SIBUNA

Eine allokationsfreie Engine zur Webabwehr durch Rechenarbeit

Zulassung gegen Rechenarbeit, Prüfung in linearer Zeit, begrenzter Zustand und Speicherung über SQL

Vikrant Rathore

Mit Unterstützung von Ronak Rathore und Kanak Rathore
Insan AI | Release 1.0.0-beta.3

ZUSAMMENFASSUNG

Web Application Firewalls und Abwehrsysteme am Netzrand leiden meist unter drei architektonischen Problemen. Das erste ist die *Kosteninversion*: Ein Absender wendet fast nichts auf, um eine Anfrage zu senden, während der Verteidiger sie parst, mit regulären Ausdrücken durchsucht, die ins Backtracking geraten können, und einen externen Speicher zu ihr befragt. Das zweite ist die *Unvorhersehbarkeit zur Laufzeit*: Heap-Allokationen auf dem Anfragepfad, Pausen der Garbage Collection und Installationen aus mehreren Containern machen das Ende der Latenzverteilung schwer beschränkbar. Das dritte ist das *betriebliche Gewicht*: Gemeinsamer Zustand für Sperren, Tokens und Limits liegt oft in Diensten, die der Betreiber neben der Firewall betreiben muss. **Sibuna** ist unser Versuch, diese Probleme mit wenigen schlichten Mechanismen in einer einzigen, eigenständigen Binärdatei zu lösen, und dieses Papier erläutert die Überlegungen hinter jedem davon. (i) *Zulassung gegen Rechenarbeit*: Ein Browser erwirbt eine Sitzung, indem er je nach Regel ein Hashcash-Rätsel oder einen Proof of Sequential Work nach Cohen-Pietrzak löst, und der Server verifiziert die Antwort in 70,2 ns bzw. 22,3 µs gegen eine Challenge, die er nie gespeichert hat. (ii) *Ein begrenzter, allokationsfreier Anfragepfad*: Ein Aho-Corasick-Automat über Byteklassen gleicht 40 Bot-Signaturen in 55,7 ns ab, 13,7-mal schneller als eine sequenzielle Suche; ein GCRA-Begrenzer entscheidet in 6,2 ns mit 16 Bytes Zustand pro Client; und eine Robin-Hood-Verbrauchsmenge verzeichnet eine Lösung in 21,0 ns. Die Prüfung auf SQL-Injection, Cross-Site-Scripting, Command Injection und Path Traversal läuft im selben Durchgang, in einer Zeit linear in der Länge der Eingabe. (iii) *Gemeinsamer Zustand ohne eigenes Konsensprotokoll*: Eine eingebettete SQLite-Datei bedient einen Knoten, eine gemeinsame PostgreSQL-Datenbank einen Cluster; beide liest ein einziger Speicherthread, auf den keine Anfrage wartet. Auf einem AMD Ryzen 9 5950X, mit dem Daemon auf vier CPUs, beantwortet Sibuna rund 200.000 Forward-Auth-Anfragen pro Sekunde (197.532 bis 211.636 je nach Arbeitslast) mit einem p99 von 305 bis 327 µs, hält sein p99.9 unter einer Last mit offener Schleife von 100.000 Anfragen pro Sekunde bei 62 µs und belegt in der Spitze etwa 16 MiB residenten Speicher. Wir legen dar, wie jede Zahl zustande kam, und führen auf, was wir nicht gemessen haben.

6,2 ns	55,7 ns	22,3 µs	16 MiB
GCRA-RATENPRÜFUNG	**BOT-ERKENNUNG**	**PoSW-VERIFIKATION**	**MAX. DAEMON-RSS**
gehashter Client | 16 Shards	Aho-Corasick | 40 Signaturen	Tiefe 13 | 16 Öffnungen	unter wrk-Last | 4 CPUs

Sibuna auf einem Kern eines AMD Ryzen 9 5950X bzw. mit vier CPUs für den Daemon, 15. Oktober 2026: benchmarks/results/primitives-latest.md und product-latest.md; der Abschnitt zur Auswertung beschreibt die Methode.

1 Prolog: Die Ökonomie einer Anfrage

Jede Anfrage, die einen Server erreicht, beansprucht seine Aufmerksamkeit. Eine Methode, ein Pfad und ein paar Header belegen nicht, dass es sich lohnt, die Anfrage zu bedienen. Eine Abwehr setzt deshalb eine Entscheidung zwischen die Anfrage und den Ursprungsserver: zulassen, ablehnen oder den Absender zuerst nachprüfbare Arbeit leisten lassen. Dieses Papier verfolgt diese Entscheidung in Sibuna von ihrer Mathematik bis zu den Bytes im Speicher.

> **Feynman über physikalische Intuition: Wer bezahlt die Arbeit?**
>
> *Stellen wir uns ein Drehkreuz vor, das sich bei der leisesten Berührung dreht. Dahinter sitzt ein Wächter, der bei jeder Drehung aufstehen, drei Kassenbücher prüfen und in der Zentrale anrufen muss. Ein Kind draußen kann den ganzen Nachmittag mühelos am Drehkreuz drehen, während der Wächter schon vor dem Mittagessen erschöpft ist. So sind viele Firewalls gebaut: Die billige Seite gehört dem Angreifer. Die Physik legt eine bessere Anordnung nahe. Wir verbinden das Drehkreuz mit einer Pumpe, sodass jeder, der hindurch will, einen Eimer Wasser in einen Tank über dem Tor heben muss. Das Heben kostet ein wenig ehrliche Mühe; der Wächter wirft nur einen Blick auf den Tank, um zu sehen, dass das Wasser gestiegen ist. Der Blick kostet den Wächter fast nichts, das Heben kostet den Besucher etwas Echtes. Der Pendler hebt einmal und geht den ganzen Tag hindurch; der Spaßvogel geht bald nach Hause.*

Die Stimmenkästen in diesem Papier sind unsere eigenen Worte, geschrieben im Geist von Richard Feynman, Donald Knuth und Leslie Lamport, deren Art, Dinge zu erklären, wir zu folgen versuchen. Sie sind keine Zitate, außer wo ein Kasten seine Quelle nennt.

Ein automatisierter Absender zahlt für jede Anfrage, die er sendet, sehr wenig. Erreicht eine Anfrage eine herkömmliche Firewall, muss der Verteidiger womöglich einen TLS-Handshake abschließen, Speicher allokieren, um die Anfrage zu kopieren und zu zerlegen, eine Liste regulärer Ausdrücke über sie laufen lassen und einen Netzwerk-Roundtrip zu einem

gemeinsamen Speicher machen, um einen Zähler zu aktualisieren. Wir nennen für diese Asymmetrie kein allgemeingültiges Verhältnis, weil es keines gibt: Es hängt vom Ursprungsserver, vom Cache und vom Angriff ab. Stattdessen macht ein einfaches Modell die Entscheidung sichtbar. Sei c_o die Arbeit des Ursprungsservers pro zugelassener Anfrage, c_g die Arbeit des Gates pro Anfrage und a der Anteil der Anfragen, die das Gate zulässt. Das Gate lohnt sich nur, wenn

$$c_g < (1 - a)\, c_o,$$

also wenn die Arbeit, die es dem Ursprungsserver erspart, seine eigenen Kosten übersteigt[1]. Daraus folgt zweierlei. Die eigenen Kosten c_g des Gates müssen klein und für jede Eingabe beschränkt sein, und die Entscheidung zuzulassen muss den Absender teuer zu stehen kommen, nicht den Server.

Sibuna kehrt die Asymmetrie um, indem es die Zulassung an einen Proof of Work knüpft. Um eine Sitzung zu erwerben, sucht ein Browser eine Nonce, deren SHA-256-Digest b führende Nullbits hat, oder berechnet einen Proof of Sequential Work. Die Suche braucht im Erwartungswert 2^b Hash-Versuche, 65.536 bei $b = 16$; der Server prüft einen einzigen Kandidaten. Die Prüfung des Servers ist zudem zustandslos: Eine Challenge auszustellen schreibt nichts, und nur eine bezahlte Lösung belegt Speicher.

Ein Proof of Work belegt weder Menschsein noch Identität noch Absicht. Ein Absender kann Rechner mieten, eine Sitzung innerhalb ihrer Lebensdauer teilen oder Arbeit auf mehrere Maschinen verteilen. Das Rätsel ändert den Preis der Zulassung; was eine zugelassene Anfrage tun darf, entscheiden weiterhin Richtlinie und Prüfung.

2 Entwurfsüberblick: Drei Entscheidungen in einem Durchgang

Sibuna hält Zulassung, Prüfung und Reputation als drei getrennte Entscheidungen auseinander und trifft sie in einem Durchgang über eine Anfrage, in einem Prozess, bevor der Ursprungsserver irgendeine Arbeit leistet. Es bietet sie in zwei Betriebsformen an. *Gate* bietet Proof-of-Work-Zulassung mit Regeln, CIDR-Reputation und lokalen Flutkontrollen. *Shield*, die Standardform, fügt eine begrenzte Prüfung der Nutzlast der Anfrage hinzu. Beide Betriebsformen laufen als Reverse Proxy vor einem Ursprungsserver oder als Forward-Auth-Endpunkt, den ein Eingangsproxy wie Caddy oder Nginx befragt. Die persistente Speicherung von Richtlinien, Reputation und Vorfallshistorie ist für beide optional:



Abb. 1 Wohin die Arbeit geht. Oben verwendet eine Abwehr, die zuerst prüft, eigene Arbeit auf jede Anfrage, die ein Absender sendet. Unten verlangt Sibuna vom Absender im Erwartungswert 2^b Hash-Versuche und verifiziert die Antwort mit einem einzigen Hash. Die Arbeitsanzeigen sind ein Modell; die Verifikationszeit ist gemessen (Hashcash-Verifikation bei 16 Bits, `benchmarks/results/primitives-latest.md`).

[1] Handbuch, Teil I, „Zulassung ist eine wirtschaftliche Entscheidung“. Die Ungleichung ist ein Kapazitätsmodell, keine Messung; sie lässt Bandbreite, Verbindungszustand und Verzögerung beim Client außer Acht.

Tabelle 1 Die beiden Betriebsformen (SID 0002; Handbuch, Teil II)

Funktion	Gate (`--gate`)	Shield (`--shield`, Standard)
Zulassung	Browser-Challenge mit Hashcash oder sequenzieller Arbeit; ein Sitzungstoken, gebunden an den Client und an die bezahlte Arbeit	Ebenso
Richtlinie	Deklarative Regeln (Pfad, User-Agent, Header, IPv4- und IPv6-CIDR, ein JA4-TLS-Fingerabdruck von einem vertrauenswürdigen Eingangsproxy) mit ALLOW, DENY, CHALLENGE und WEIGH; Bot-Signaturen; Reputation	Ebenso
Flutkontrollen	GCRA-Ratenlimits, Honeypot, temporäre Sperren	Ebenso
Prüfung der Nutzlast	Deaktiviert	SQL-Injection, Cross-Site-Scripting, Command Injection und Path Traversal, über die ersten 8 KiB des Bodys
Optionaler Zustand	Richtlinien und Reputation in SQLite oder gemeinsam über PostgreSQL	Zusätzlich Vorfallsforensik: Volltextsuche und Gruppierung zu Kampagnen

lokal auf einem Knoten oder gemeinsam für einen Cluster[2].

2.1 Die Entwurfsentscheidungen

Die meisten der folgenden Ideen stammen aus der Forschungsliteratur oder aus früheren Systemen; Teil II des Handbuchs zeichnet ihre Herkunft nach. Wir haben versucht, sie in einem einzigen begrenzten Anfragepfad zu verbinden, jeden Schritt billig zu halten und das Ergebnis ehrlich zu messen. Wir führen die Entscheidungen mit ihrer jeweiligen Begründung auf.

1. *Challenges, deren Ausstellung den Server nichts kostet.* Eine Challenge ist ein selbstauthentifizierender Datensatz, versiegelt mit einem Hash mit Schlüssel; ihre Ausstellung schreibt also keinen Zustand. Nur eine bezahlte Lösung belegt Speicher, und sie verschwindet, wenn die Challenge abläuft (Algorithmen und ihre Garantien).
2. *Zwei Stufen von Arbeit, je Regel gewählt.* Neben Hashcash bietet Sibuna den Proof of Sequential Work nach Cohen-Pietrzak an, dessen Korrektheit sogar gegen Quanten-Beweiser bewiesen ist und dessen Nachweis sich in Mikrosekunden verifizieren lässt. Eine Checkout-Route kann 20 Arbeitsbits sequenzieller Arbeit verlangen, während ein Blog 16 Bits Hashcash verlangt (Proofs of Work: Parallele Suche und sequenzielle Arbeit).
3. *Sitzungen, die sich an die bezahlte Arbeit erinnern.* Ein Sitzungstoken hält das Verfahren und die Arbeitsbits fest, die sein Inhaber gelöst hat. Es öffnet nur Routen, die nicht mehr verlangen; eine billige Sitzung kann also keine teure Route öffnen.
4. *Symmetrische Tokens.* Sitzungen werden mit BLAKE3 mit Schlüssel versiegelt. Das verifiziert sich in 155,8 ns, mit etwa 260-mal weniger Arbeit als eine Ed25519-Signatur, und eine Quantensuche bringt dagegen nur eine quadratische Beschleunigung.
5. *Eine Prüfung, die keine Sitzung umgehen kann.* Zulassung und Prüfung laufen im selben Durchgang, und eine gültige Sitzung erfüllt nur eine Challenge: Eine Anfrage mit Sitzung und `q=' OR 1=1--` wird trotzdem abgewiesen.
6. *Strukturbewusste Prüfung in linearer Zeit.* Ein Byteklassen-Scan, ein Durchlauf des Automaten und kleine strukturelle Tokenizer entscheiden in einer Zeit linear in der Eingabe, ohne reguläre Ausdrücke, die ins Backtracking geraten könnten.
7. *Begrenzter Zustand mit schlüsselbasierter Platzierung.* Jede Tabelle hat eine feste Kapazität und ein festgelegtes Verhalten, wenn sie voll ist, und Schlüssel werden mit einem Hash mit Schlüssel platziert, sodass ein Angreifer Kollisionen nicht gezielt gegen ein Opfer richten kann (Hashing mit Schlüssel und die Sicherheitsprüfung).
8. *Keine Allokation pro Anfrage und Richtlinienupdates ohne Pausen.* Eine Verbindung bedient jede Entscheidung aus einem einzigen festen Puffer, und eine neue Richtlinien-Engine wird veröffentlicht, ohne dass eine Anfrage oder ein Schreiber wartet (Mechanical Sympathy: Keine Allokation und begrenzter Zustand).
9. *Eine Schwierigkeit, die der Last folgt.* Oberhalb einer Basis von 50 verifizierten Lösungen pro Sekunde fügt jede Verdopplung der Rate ein Arbeitsbit hinzu, bis zu sechs; eine Flut zahlt also exponentiell mehr, während ruhiger Datenverkehr die Grundkosten behält.
10. *Kampagnen aus Vorfällen.* Abgelehnte Anfragen werden mit einem kleinen Embedding aufgezeichnet, und ähnliche Vorfälle werden zu Kampagnen gruppiert, die ein Betreiber in der Konsole untersuchen kann.

[2]SID 0002, „Decision“; Handbuch, Teil II, „Drei Ideen in einer Binärdatei“; Handbuch, Teil VI, „Die deklarative Richtlinien-Engine“. Das Beispiel je Regel, das Anforderungsticket und die Sitzung mit Arbeitsstufe stammen aus Teil IV des Handbuchs, die adaptive Basis aus Teil IV des Handbuchs und SID 0006 (Revisionsnotiz, Teil H).

3 Mechanical Sympathy: Keine Allokation und begrenzter Zustand

Knuth über mechanische Präzision: Cache-Zeilen und konkrete Mathematik

Wer in der inneren Schleife eines Echtzeitsystems einen Allzweck-Allokator aufruft, hat einen Teil des Entwurfs einem Fremden überlassen. Der Allokator mag im Mittel schnell sein, aber seine Kosten hängen von der Geschichte des Heaps ab, und diese Geschichte schreibt, wer auch immer die Anfragen sendet. Bevor wir irgendetwas optimieren, sollten wir zählen: wie viele Bytes jeder Schritt berührt, wie viele Cache-Zeilen, wie viele abhängige Ladezugriffe. Dann sollten wir messen, denn auch Zählen kann irren. Ein Programm ist elegant, wenn jedes Byte, das es braucht, einen bekannten Platz hat, bevor der erste Socket geöffnet wird, und wenn jede Schleife genau die Arbeit leistet, die ihre Mathematik verlangt.

3.1 Die Invariante der Allokationsfreiheit

Jede dynamische Allokation auf einem Anfragepfad ist ein Hebel für Denial of Service: Fragmentierung über Tage, Konkurrenz um die Sperren des Allokators und Verstärkung durch Anfragen, die gezielt Allokationen auslösen. Sibunas Regel lautet: Klassifikation, Verifikation und die Übergabe an den Proxy laufen ohne Heap-Allokation. Eine Verbindung besitzt für die Lebensdauer ihres Sockets einen Lesepuffer von 64 KiB und einen Schreibpuffer von 16 KiB sowie einen Proxy-Arbeitsbereich von 40 KiB, der einmal beim Start der Verbindung allokiert wird. Alles andere ist ein Slice in diese Puffer: Der Parser findet Methode, Pfad, Query, Header und Cookies einer Anfrage im Lesepuffer und kopiert nichts davon[3]. Das Typsystem der Implementierung erzwingt, dass eine geparste Anfrage die Bytes, die sie beschreibt, nicht überleben kann; der Puffer kann also nicht neu gefüllt werden, solange eine Anfrage noch aus ihm liest.

Die Regel ist ein Vertrag des Anfragepfads, keine Aussage über den ganzen Prozess. Threaderzeugung, Start und Speicherung allokieren. Den Vertrag prüfen Tests, die Allokationen zählen: Der Test des Daemons im eingeschwungenen Zustand zählt seine Allokationen, und der des Proxys findet über 500 weitergeleitete Keep-alive-Anfragen keine einzige[4].

3.2 Begrenzte Threads, wiederverwendet

Jede angenommene Verbindung wird von einem eigenen Thread bedient; ein langsamer Client verzögert also nur sich selbst, und der Anfragepfad bleibt synchron. Wir haben eine Event-Loop erwogen und das Thread-Modell beibehalten: Die Geschwindigkeit des Anfragepfads kommt von einem Zero-Copy-Parser, einem festen Puffer pro Verbindung und allokationsfreien Entscheidungen, und nichts davon verbessert eine Event-Loop[5]. Die Zahl der Verbindungsthreads ist durch `--max-connections` begrenzt (standardmäßig 1.024); darüber hinaus antwortet die Accept-Schleife mit `503`, ohne etwas zu starten. Eine Peer-Adresse eines Sockets darf standardmäßig höchstens 64 Verbindungen halten, und ein IPv6-Peer wird mit seinem /64-Netz gezählt.

Threads stammen aus einer festen Tabelle von Slots, die sich joinen lassen. Ein beendeter Thread behält seinen Stack, bis er gejoint wird; deshalb joint eine neue Aufgabe den *niedrigsten* beendeten Slot und übernimmt ihn, statt einen Cursor weiterzurücken. Der residente Speicher folgt dann den offenen Verbindungen, nicht den bedienten. Ein Thread, der seine Verbindung beendet hat, parkt und übernimmt die nächste. Der Abschnitt Was die Messung verändert hat erzählt, wie uns eine Messung diese letzte Regel gelehrt hat.

3.3 Die Anfrage-Pipeline

Eine Anfrage durchläuft die folgenden Prüfungen in fester Reihenfolge. Die billigen Prüfungen kommen zuerst, und jede Entscheidung, die die Anfrage beendet, fällt, bevor eine teurere Stufe läuft.

Eine gültige Sitzung erfüllt eine Challenge; sie hebt nie eine Ablehnung oder einen WAF-Befund auf. Der Proxy wiederum liest den Antwortkopf des Ursprungsservers, bestimmt dessen Rahmung (kein Body, eine Länge, Chunked-Kodierung oder bis zum Schließen), leitet den Body exakt weiter und gibt Sockets zum Ursprungsserver, die an einer sauberen Grenze stehen, an einen in Streifen geteilten Pool mit 256 Plätzen zurück. So bleibt eine weitergeleitete Client-Verbindung offen, und zum Ursprungsserver wird nicht für jede Anfrage neu verbunden.

3.4 Richtlinien veröffentlichen, ohne Anfragen anzuhalten

Die Richtlinien-Engine wird abseits des Anfragepfads neu gebaut, wenn sich Richtlinien ändern, und Anfragen dürfen nie eine halb gebaute Engine sehen. Der Speicherthread baut eine vollständige Engine und veröffentlicht sie referenzgezählt in einem Slot mit Generationszähler. Jede Verbindung hält einen Cache: Sie lädt die Generation mit einem einzigen Acquire-Load und nimmt erst nach einer Veröffentlichung eine neue Referenz. Eine alte Engine wird freigegeben, sobald der letzte Cache, der sie hält, sie loslässt, und kein Schreiber wartet je darauf, dass die Leser fertig werden. Unter Konkurrenz ist das gecachte Lesen mehr als 400-mal billiger als das Nehmen einer Lese-Schreib-Sperre. Die Engine selbst hält etwa 200 KiB kompakter Tabellen (209.052 Bytes), mit Platz für 128 Regeln und 8.192 Trie-Knoten.

[3] Handbuch, Teil V, „Ein Thread pro Verbindung, begrenzt“ und „Ein Puffer pro Verbindung“.
[4] Handbuch, Teil V, „Die Verbindungsschleife“; `cmd/sibuna/tests/steady_state.rs` und `crates/net/tests/steady_state.rs`.
[5] SID 0010, „Concurrency model“ und „Rationale and alternatives“; SID 0002, „Request contract“ und seine Revisionsnotizen (Verbindungen pro Adresse, IPv6-Präfix, der in Streifen geteilte Pool zum Ursprungsserver).

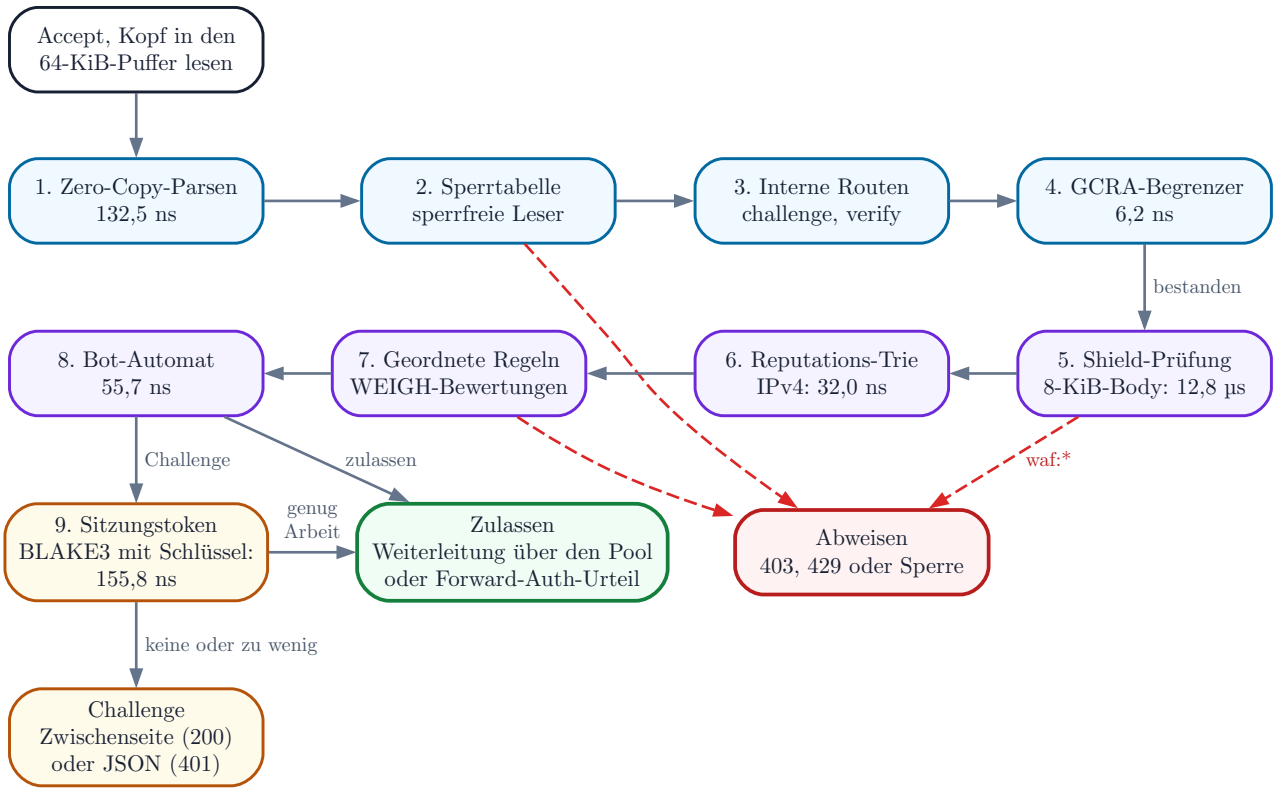


Abb. 2 Die Anfrage-Pipeline mit der Medianzeit jeder Primitive auf einem Kern, sofern die Primitive-Suite sie misst (`benchmarks/results/primitives-latest.md`). Die Stufen laufen in der nummerierten Reihenfolge; gestrichelte Pfeile zeigen frühe Abweisungen (der GCRA-Begrenzer und der Reputations-Trie weisen auf dieselbe Weise ab). Die gesamte Klassifikation einer Browseranfrage dauert im Gate-Profil 75,9 ns und im Shield-Profil 790,8 ns.

3.5 Sperren in Shards

Die Tabellen, die sich die Verbindungsthreads teilen (die Verbrauchsmenge, der Begrenzer, die Sperrtabelle), sind über einen Hash des Schlüssels in 16 Shards aufgeteilt, und jeder Shard liegt auf einer eigenen Cache-Zeile. Ihre kritischen Abschnitte sind nur einige Dutzend Instruktionen lang; deshalb haben wir einen Spinlock gegen einen adaptiven Mutex gemessen, der kurz spinnt und dann schläft. Ein Spinlock spart etwa eine Nanosekunde, solange niemand konkurriert. Ein Edge-Server betreibt aber mehr Threads als CPUs, und ein Spinlock, dessen Halter verdrängt wird, lässt jeden Wartenden seine Zeitscheibe verbrennen. Unter Konkurrenz gewann der Mutex, also haben wir ihn behalten.

4 Algorithmen und ihre Garantien

Für jede Primitive auf dem Anfragepfad gibt es eine Aussage darüber, was sie kostet und was sie garantiert. Drei davon geben wir als Sätze an, mit einer knappen Begründung; SID 0006 und die Teile III bis VI des Sibuna-Handbuchs enthalten die vollständigen Beweise, die Definitionen und die Literaturangaben.

Tabelle 2 Kosten eines Engine-Lesezugriffs mit 1 und 16 Threads, die in einer Schleife lesen, auf einem Ryzen 9 5950X (Handbuch, Teil V, Oktober 2026)

Leser	1 Thread	16 Threads
Lese-Schreib-Sperre, nur Lese-Guard	4,2 ns	752 ns
Lese-Schreib-Sperre und eine neue Referenz	7,6 ns	833 ns
Generationsprüfung mit gecachter Referenz	0,8 ns	1,8 ns

Tabelle 3 Durchsatz der Shard-Sperren in Millionen Operationen pro Sekunde auf einem Ryzen 9 5950X (Handbuch, Teil V; SID 0010)

Arbeitslast	Adaptiver Mutex	Spinlock
Verbrauchsmenge, 16 Threads	10,8	8,6
Verbrauchsmenge, 64 Threads	11,2	6,9
GCRA auf sechs Hotspot-Adressen, 16 Threads	44,9	19,3
GCRA auf sechs Hotspot-Adressen, 64 Threads	37,4	6,5

Satz 1 (Prüfung in linearer Zeit mit einem Aho-Corasick-Automaten über Byteklassen). Sei $\mathcal{P}$ eine Menge von Mustern der Gesamtlänge m, deren Bytes in $c \leq 256$ Klassen fallen, und habe ein Text die Länge M. Sibuna baut den Automaten in $O(mc)$ Zeit und Speicher und meldet in $O(M)$ Zeit, ob irgendein Muster im Text vorkommt, mit einem abhängigen Tabellenzugriff pro Eingabebyte, unabhängig von der Zahl der Muster. Keine Eingabe kann ihn zum Backtracking bringen.

Beweisskizze. Der Automat von Aho und Corasick durchläuft einen Text einmal, nachdem er einen Trie mit Fehlerkanten gebaut hat. Sibuna faltet die Fehlerkanten beim Bauen in die Übergangstabelle; jedes Eingabebyte ist daher ein Übergang, und keiner Fehlerkette wird je gefolgt. Eine Byteklasse ist ein Byte, das in einem Muster vorkommt, wobei Groß- und Kleinschreibung eines Buchstabens eine Klasse bilden, oder die eine Klasse aller übrigen Bytes; eine Zeile der Tabelle hat deshalb c Einträge statt 256. Trefferzustände werden nach allen anderen Zuständen nummeriert, sodass das Erkennen eines Treffers ein einziger Vergleich ist.

Messung. Bei 40 Bot-Signaturen braucht der Automat 55,7 ns gegenüber 763,0 ns für eine sequenzielle Teilzeichenkettensuche mit denselben Mustern und derselben Semantik „irgendein Treffer“, ist also 13,7-mal schneller. Mit Byteklassen belegt die gesamte Tabelle der Bot-Signaturen 42.244 Bytes[6].

□

Satz 2 (Die GCRA-Intervallschranke). Wir legen ein Limit von N Anfragen pro Fenster W fest, mit dem Emissionsintervall $T = \max(1, \lceil W/N \rceil)$ und der Burst-Toleranz $\tau = (N-1)T$. Ein Client führt eine theoretische Ankunftszeit TAT. Eine Ankunft zur Zeit t ist genau dann konform, wenn $\mathrm{TAT} \leq t + \tau$ gilt; danach wird $\mathrm{TAT} \leftarrow \max(\mathrm{TAT}, t) + T$ gesetzt. Dann enthält jedes Intervall der Länge L, das mit einer konformen Ankunft beginnt, höchstens $N + \lfloor L/T \rfloor$ konforme Ankünfte.

Beweisskizze. Wir nummerieren die konformen Ankünfte $1, ..., k$ zu den Zeiten $t_1 \leq ... \leq t_k$. Jede konforme Ankunft erhöht die TAT um mindestens T, also gilt nach der j-ten $\mathrm{TAT}_j \geq t_1 + jT$. Ankunft k war konform, also gilt $\mathrm{TAT}_{k-1} \leq t_k + \tau \leq t_1 + L + \tau$. Zusammen ergibt das $(k-1)T \leq L + (N-1)T$, also $k \leq N + L/T$, und k ist eine ganze Zahl.

Implementierung. Der Zustand eines Clients ist eine einzige 16-Byte-Zelle, ohne Verlauf pro Anfrage und ohne Randeffekte fester Fenster. Die 16 Shards teilen sich `--rate-table-cells` Zellen (standardmäßig 262.144, 4 MiB), platziert über einen Hash mit Schlüssel, und eine geleerte Zelle wird an Ort und Stelle zurückgewonnen, ohne Aufräumthread. Eine Prüfung für einen Client, dessen Hash der Daemon bereits berechnet hat, dauert 6,2 ns[7].

□

Satz 3 (Zustand wächst nur mit bezahlter Arbeit). Sei S die Menge der Challenge-Tags, die sich der Daemon merkt. Ein Tag gelangt erst in S, nachdem ein gültiger Nachweis für seine Challenge akzeptiert wurde, und verlässt sie wieder, wenn die Challenge abläuft. Beschränkt r die Rate akzeptierter verschiedener Lösungen über eine Lebensdauer einer Challenge, dann gilt $\mid S \mid \leq r \cdot \mathrm{TTL}$. Eine Challenge, die abgerufen, aber nicht gelöst wurde, belegt keinen Speicher.

Beweisskizze. Eine Challenge-Kennung ist ein selbstauthentifizierender Datensatz: Version, Algorithmus, Schwierigkeit, Zahl der Öffnungen, Ausstellungszeit, der schlüsselbasierte Fingerabdruck des Clients, eine von einer Pseudozufallsfunktion abgeleitete Nonce und der Hash der Regel, die sie verlangt hat; das sind 36 Bytes, gefolgt von einem 16-Byte-Tag aus BLAKE3 mit Schlüssel, 70 Zeichen in URL-sicherem Base64. Ihre Ausstellung schreibt nichts. Die Verifikation prüft zuerst das Tag in konstanter Zeit, dann das Alter und die Bindung an den Client, dann den Nachweis und fügt erst danach das Tag in die Verbrauchsmenge ein. Modelliert man BLAKE3 im Schlüsselmodus als Pseudozufallsfunktion, dann fälscht ein Gegner mit q Verifikationsanfragen ein Tag höchstens mit der Wahrscheinlichkeit $q \cdot 2^{-128}$ plus dem PRF-Vorteil.

Implementierung. Die Verbrauchsmenge besteht aus 16 Shards mit Robin-Hood-Open-Addressing. Ein abgelaufener Eintrag darf nur von einem Schlüssel an Ort und Stelle überschrieben werden, der mindestens so weit von seinem Heimatslot entfernt ist; sonst könnte eine spätere Suche zu früh enden und ein lebendes Tag verfehlen. Mit dieser Regel ist kein Aufräumthread nötig, und eine abgewiesene Einfügung verdrängt nie ein lebendes Tag. Verbrauchen und Nachschlagen dauern zusammen 21,0 ns[8].

□

[6] `benchmarks/results/primitives-latest.md` (55,7 und 763,0 ns); die Tabellengröße aus `primitives-latest.json`, wie in Teil VIII des Handbuchs, „Speicherbedarf“, angegeben; SID 0006, Teil F, und seine Revisionsnotiz zu Byteklassen.

[7] SID 0006, Teil D (Definition 22, Satz 23); SID 0004, Revisionsnotizen (schlüsselbasierte Platzierung, `--rate-table-cells`); `benchmarks/results/primitives-latest.md` (`gcra_check_hashed`).

[8] Handbuch, Teil III (Lemma 3 und Lemma 4, Übertragungsformate); Handbuch, Teil V, „Die Robin-Hood-Verbrauchsmenge“; SID 0006, Teil E (Lemma 28); `benchmarks/results/primitives-latest.md` (`robin_hood_spend_and_lookup`).

4.1 Die semantische Firewall

Shield prüft den Pfad, die Query, den User-Agent, die nicht strukturellen Header und die ersten 8 KiB des Bodys. Seine erste Version hat uns eine Lektion erteilt. Sie verglich eine flache Liste von Teilzeichenketten mit jedem Header; die Liste enthielt `/*`, und jeder Browser sendet `Accept: text/html,...,*/*;q=0.8`. Jeder echte Browser wurde als SQL-Injection eingestuft. Seitdem darf ein Detektor nur auf *Struktur* anschlagen, und ein Test schickt eine vollständige Chrome-Anfrage durch die Engine und verlangt, dass sie unbeanstandet herauskommt[9].

Die Prüfung hat vier Stufen, jede linear in der Länge des Felds.

1. *Eine Vorabsuche über Byteklassen.* Ein Durchlauf mit einer Tabelle von 256 Einträgen hält fest, welche Byteklassen vorkommen: Anführungszeichen, `=`, `<`, Shell-Trennzeichen, `%`, NUL und die Auslöser der Kanonisierung. Jeder spätere Detektor fragt diese Klassen ab, statt erneut zu suchen; ein Feld ohne Anführungszeichen und ohne Gleichheitszeichen gelangt also nie in den SQL-Tokenizer.
2. *Ein markierter Signaturautomat.* Alle starken Signaturen aller Kategorien liegen in einem einzigen Automaten (Satz 1). Ein Treffer ist eine endgültige Ablehnung mit dem Namen `waf:sqli`, `waf:xss`, `waf:path-traversal` oder `waf:rce`.
3. *Strukturelle Tokenizer.* Der SQL-Tokenizer schlägt jedes kurze Wort in einer Schlüsselworttabelle nach und prüft nach `or` und `and` auf Tautologien der Form `literal = literal`; andernfalls ist ein Anführungszeichen Pflicht, und die Indizien müssen eine kleine Punktzahl erreichen. Der Satz „it's a group order from the shop“ erreicht 3 Punkte und kommt durch. Cross-Site-Scripting verlangt einen Tag, der Skript aufnehmen kann, zusammen mit einem Event-Handler-Attribut an Attributposition, und Command Injection verlangt ein Trennzeichen, gefolgt von einem Befehlsnamen, der keine Zuweisung ist; `a=1&id=5` ist also ein Parameter, kein Befehl.
4. *Kanonisierung.* Ein Feld mit Prozent-Escapes, Kommentaranfängen, JSON- oder HTML-Escapes oder zusammenfassbarem Leerraum wird dekodiert und erneut geprüft; so fallen `1%27/**/UnIoN/**/SeLeCt` und `%252e%252e/` auf die rohen Signaturen zusammen.

Ein 8-KiB-Body wird in 12.784,3 ns durchsucht, etwa 1,56 ns pro Byte. Ein Betreiber kann eine Kategorie im Audit-Modus betreiben, statt sie durchzusetzen; ihre Befunde werden dann protokolliert, aber nicht befolgt. Ein Feld muss dann je Kategorie beurteilt werden, damit ein auditierter Befund keinen durchgesetzten verdecken kann. Statt das Feld einmal pro Kategorie zu durchsuchen, durchsucht Sibuna es einmal nach der Menge der Kategorien, deren Signaturen vorkommen, und sucht dann nur unter diesen nach den gewünschten Kategorien; derselbe 8-KiB-Body mit SQL-Injection im Audit-Modus dauert 15.485,4 ns[10]. Shield bleibt eine begrenzte heuristische Prüfung, kein vollständiger SQL- oder HTML-Parser, und Bytes jenseits der Prüfgrenze werden nicht geprüft.

4.2 TLS-Fingerabdrücke vom Eingangsproxy

Ein User-Agent ist eine Zeichenkette, die jeder Client nach Belieben schreiben kann. Ein JA4-Fingerabdruck ist schwerer zu fälschen. Er fasst zusammen, wie ein Client eine TLS-Verbindung öffnet: Protokoll, Version, Cipher Suites, Erweiterungen und den ersten ALPN-Wert. Die TLS-Bibliothek des Clients legt diese fest, bevor das erste HTTP-Byte gesendet wird. Nur die Komponente, die TLS terminiert, sieht den Handshake; nur sie kann also JA4 berechnen. Sibuna terminiert kein eingehendes TLS und berechnet daher auch kein JA4. Es liest den Fingerabdruck, den ein vertrauenswürdiger Eingangsproxy berechnet und in einem Anfrage-Header weitergeleitet hat, dessen Name mit `--ja4-header` festgelegt wird[11].

- *Vertrauen.* Der Header wird nur gelesen, wenn weitergeleiteten Headern vertraut wird, nach derselben Entscheidung, die auch für `X-Forwarded-For` gilt. Von einem nicht vertrauenswürdigen Peer wird er ignoriert, und im Reverse-Proxy-Modus wird jede Kopie entfernt, bevor die Anfrage den Ursprungsserver erreicht.
- *Validierung.* Ein Wert wird nur in der 36 Bytes langen JA4-Form akzeptiert, und nur wenn genau ein Feld dieses Namens vorhanden ist. Ein fehlerhafter oder wiederholter Wert gilt als fehlend und wird gezählt.
- *Richtlinie.* Eine Regel passt auf einen ganzen Fingerabdruck oder auf ein Präfix, gefolgt von `*`; eine Anfrage ohne Fingerabdruck passt nie auf eine solche Regel. Der Abgleich ist ein Bytevergleich, und der warme Anfragepfad allokiert weiterhin keinen Heap-Speicher.
- *Aufzeichnungen.* Vorfälle speichern den Fingerabdruck ihrer Anfrage, und die Konsole zeigt ihn in den Belegen eines Vorfalls und filtert Ereignisse danach. Sperren und Reputation bleiben an die Adresse gebunden, weil viele legitime Clients dieselbe TLS-Bibliothek verwenden.

4.3 Hashing mit Schlüssel und die Sicherheitsprüfung

Eine Prüfung des Anfragepfads aus Angreifersicht fand im Oktober 2026 Schwächen in Mechanismen, die wir für erledigt gehalten hatten. Wir haben jede einzelne behoben und die Korrektur im Entwurfsdokument des jeweiligen Mechanismus festgehalten[12].

[9] SID 0004, „Revision note (2026-09-07)“ und „Verification“; Handbuch, Teil VI, „Eine Lektion über Fehlalarme“.

[10] `benchmarks/results/primitives-latest.md` (`8kb_body_semantic_scan`, `8kb_body_sqli_audit_other_enforce`); SID 0004, Revisionsnotiz „inspection modes“; Handbuch, Teil VIII, „WAF-Prüfung“.

[11] SID 0014, „Specification“ und „Security and operational considerations“; Handbuch, Teil IX, „JA4-Fingerabdrücke vom Eingangsproxy“; `.github/release-notes.txt`.

[12] SID 0002, Revisionsnotizen (weitergeleitete Identität; Cookie-Werte); SID 0004, Revisionsnotizen (schlüsselbasierte Platzierung, Sperrtabelle, gemessene Kosten, mit Nullen initialisierte Tabellen); SID 0006, Revisionsnotizen (Kapazität der Verbrauchsmenge, Eingabe des adaptiven Reglers); SID 0010, Review-Verlauf (Sicherheitsprüfung, 2026-10-15); `benchmarks/results/primitives-latest.md` (`gcra_check` und `gcra_check_hashed`).

- *Vorhersagbare Platzierung in Tabellen.* Der Begrenzer und die Sperrtabelle platzierten Schlüssel mit einem Hash unter Seeds, die im Quellcode standen; jeder konnte also Adressen berechnen, die sich das Suchfenster eines Opfers teilen. Die Platzierung nutzt jetzt SipHash-1-3 unter einem Schlüssel aus der Zufallsquelle des Betriebssystems, und jede Tabelle leitet ihren eigenen Schlüssel ab. Der Daemon hasht den Client einmal pro Anfrage (8,7 ns) und reicht den Hash an jede Tabelle weiter; der Speicherpfad einer Anfrage sank von 43 auf 26 ns.
- *Verdrängung von Sperren.* Eine wirksame Sperre wird nie verdrängt. Wenn jeder Slot, den ein Schlüssel einnehmen kann, eine lebende Sperre hält, wird die neue Sperre abgewiesen und gezählt; ein gesperrter Client kann also seine eigene Sperre nicht verdrängen.
- *Weitergeleitete Identität.* Den linken Eintrag von `X-Forwarded-For` kann der Client schreiben. Sibuna nimmt den Eintrag, der `--trusted-hops` Positionen von rechts steht, also den, den der nächstgelegene vertrauenswürdige Proxy angehängt hat.
- *Cookie Tossing.* Eine Anfrage kann mehrere Werte des Sitzungscookies mitführen; der Daemon probiert bis zu acht, damit ein von einer Schwesterdomain gesetztes Cookie einen Besucher nicht in einer Challenge-Schleife festhalten kann.
- *Kapazität der Verbrauchsmenge.* Eine Tabelle mit 65.536 Tags war bei etwa 220 verifizierten Lösungen pro Sekunde innerhalb einer fünfminütigen Lebensdauer der Challenges erschöpft; danach wurden ehrliche Lösungen abgewiesen. Die Kapazität ist jetzt konfigurierbar, standardmäßig 1.048.576 Tags (24 MiB).
- *Adaptive Schwierigkeit.* Der Regler, der die Arbeit unter Last erhöht, zählt jetzt verifizierte Lösungen statt ausgestellter Challenges, die einen Angreifer, der flutet, nichts kosten.

Die schlüsselbasierte Platzierung hat ihren Preis, und wir zahlen ihn einmal pro Anfrage statt einmal pro Tabelle. Eine Begrenzerprüfung, die die rohen Client-Bytes selbst hasht, dauert 12,8 ns; die Prüfung für einen bereits gehashten Client, der Weg, den der Daemon nutzt, dauert 6,2 ns. Weil der Begrenzer und die Verbrauchsmenge mit Nullen initialisiert allokiert werden, sind ungenutzte Tabellen nicht resident; ihre konfigurierte Kapazität kostet also keinen Speicher, solange sie nicht genutzt wird.

5 Proofs of Work: Parallele Suche und sequenzielle Arbeit

5.1 Die Hashcash-Stufe

Die erste Stufe ist Hashcash mit Bit-Granularität. Für eine Challenge-Zeichenkette C und die Schwierigkeit b sucht der Client eine Nonce N, sodass SHA-256 $(C \parallel \texttt{":"} \parallel \mathrm{dec}\,(N))$ b führende Nullbits hat. Die dezimale Darstellung der Nonce hält das Übertragungsformat zwischen dem WebAssembly-Löser, dem JavaScript-Fallback und dem nativen Verifizierer identisch. Für ideale Digests ist die Zahl der Versuche K geometrisch verteilt mit $p = 2^{-b}$:

$$P(K > k) = (1-p)^k, \qquad E[K] = 2^b, \qquad \mathrm{median}\,(K) \approx 2^b \ln 2.$$

Die Suche hat zwei Grenzen. Ihre Varianz ist groß; eine langsame Lösung bedeutet also keinen defekten Worker. Und unabhängige Versuche laufen parallel; ein Absender mit mehr Kernen ist also schneller fertig, obwohl die erwartete Gesamtarbeit gleich bleibt[13].

5.2 Die Stufe der sequenziellen Arbeit

Die zweite Stufe folgt dem Proof of Sequential Work von Cohen und Pietrzak (EUROCRYPT 2018). Seine Labels bilden einen vollständigen Binärbaum der Tiefe n. Ein innerer Knoten hasht seine beiden Kinder. Ein Blatt hasht die *linken Geschwister seiner Vorfahren*, überall dort, wo sein Wurzelpfad nach rechts abbog. Diese Kanten zu linken Geschwistern machen die Berechnung sequenziell: Ein Blatt kann erst beschriftet werden, wenn jeder Teilbaum links davon fertig ist; das Wurzellabel hängt also der Reihe nach von allen $2^{n+1} - 1$ Labels ab. Nachdem der Beweiser die Wurzel φ berechnet hat, leitet er per Fiat-Shamir t Blätter ab und sendet für jedes das Blattlabel und die n Geschwisterlabels entlang seines Pfads. Der Verifizierer berechnet jedes geöffnete Blatt neu und hasht bis hinauf zu φ, mit Kosten von $O(tn)$ Labelberechnungen in einem festen Arbeitsbereich.

Die Parameter sind so gewählt, dass die Zahlen klein bleiben. Arbeitsbits werden über $n = b - 3$ auf die Tiefe abgebildet, begrenzt auf $[4, 24]$. Ein Nachweis hat $32(1 + t(n+1))$ Bytes, bei Tiefe 13 mit 16 Öffnungen also $32(1 + 16 \times 14) = 7200$ Bytes. Der Beweiser hält nur die oberen Ebenen des Baums vor und berechnet den Teilbaum, den er öffnen muss, neu; sein gesamter Arbeitsbereich, den Nachweis eingeschlossen, bleibt so unter 100 KB. Ein Browser-Tab und die nativen Tests verwenden denselben Beweiser[14]. Blocki, Lee und Zhou (ITC 2021) haben die Konstruktion im Quanten-Random-Oracle-Modell als korrekt gegen Quanten-Beweiser bewiesen.

Wir sollten vorsichtig sein, was damit belegt ist. Die Sicherheitsergebnisse gelten unter den Random-Oracle-Annahmen der Arbeiten, für die Konstruktionen und Schranken, die sie angeben. Sie sind keine unabhängige Prüfung unseres konkreten SHA-256-Protokolls, seiner Parameterabbildung oder seines Zeitverhaltens im Browser. Die elementare Stichprobenschranke $(1-\alpha)^t$, die Wahrscheinlichkeit, dass t unabhängige Öffnungen alle einen festen schlechten Anteil

[13] Handbuch, Teil III, „Hashcash: Die Mathematik von Stufe eins“; Handbuch, Teil I, „Hash-Suche als Zufallsvariable“.
[14] Handbuch, Teil III, „Nachweis sequenzieller Arbeit: Stufe zwei“, „Speicher des Beweisers“ und „Parameter sind keine Zeitangaben“; SID 0006, Teil B und sein Implementierungsreview vom 2026-09-07.

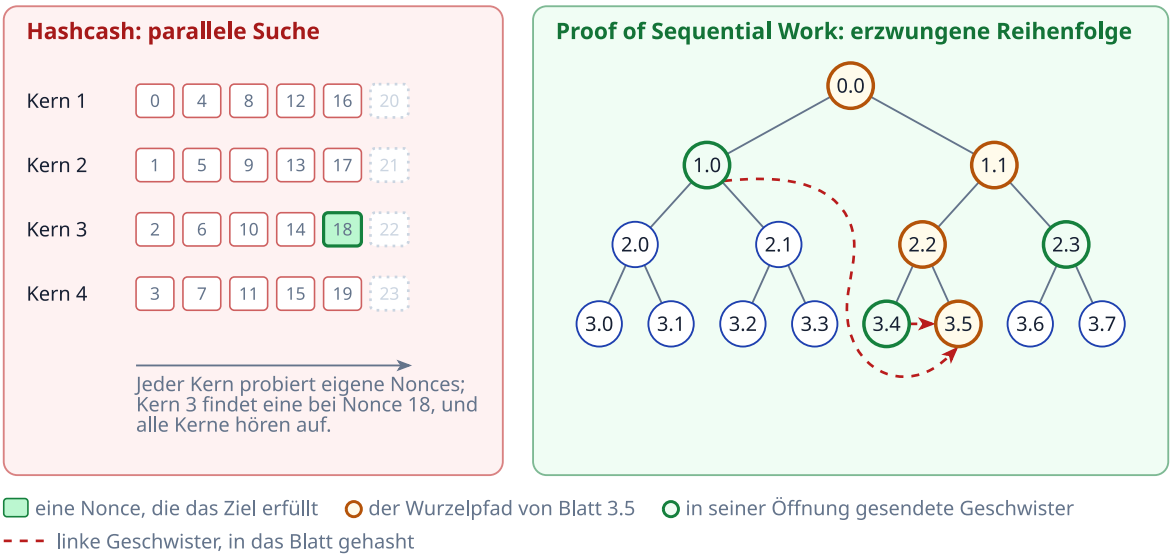


Abb. 3 Parallele Suche gegen sequenzielle Arbeit. Links Hashcash: unabhängige Nonce-Versuche, sodass M Kerne etwa M-mal schneller fertig sind. Rechts der Baum nach Cohen-Pietrzak mit Tiefe 3: Die gestrichelten Kanten zu linken Geschwistern lassen das Blatt 3.5 auf die Teilbäume links davon warten, und eine Öffnung des Blatts 3.5 sendet sein Label und die drei grünen Geschwister (nach Handbuch, Teil III).

α verfehlen, erklärt, warum mehr Öffnungen helfen; für sich genommen ist sie kein Beweis gegen einen Beweiser, der viele Wurzeln durchprobiert. Wir behaupten nicht bedingungslos, dass Betrug nie billiger ist.

5.3 Der Rundlauf und die Sitzung

Eine Navigation ohne gültige Sitzung erhält die Zwischenseite, die ein *Anforderungsticket* trägt: die Challenge, die diese Entscheidung verlangt (Algorithmus, Arbeitsbits, Öffnungen), und den Hash der Regel, versiegelt für diesen Client und gültig für eine Lebensdauer einer Challenge. Die Seite holt ihre Challenge mit dem Ticket; die Arbeit ist also durch die Entscheidung festgelegt, die sie verlangt hat. Ein Web Worker löst die Challenge mit dem WebAssembly-Beweiser oder, wo WebAssembly nicht verfügbar ist, mit einem JavaScript-Fallback und sendet die Antwort. Bei Erfolg setzt der Server ein Sitzungscookie aus 75 Zeichen, über einen schlüsselbasierten Fingerabdruck an Adresse und User-Agent des Clients gebunden, und die Seite lädt neu. Der Beweiser im Browser wird aus demselben Quellcode kompiliert wie der Verifizierer des Servers; die beiden können also nicht auseinanderlaufen[15].

5.4 Was die Verifikation kostet

Beide Stufen sind durch SHA-256 bestimmt, und Sibuna wählt zur Laufzeit die SHA-Instruktionen des Prozessors, wo es sie gibt; eine einzige Binärdatei ist also auf jeder Maschine schnell, die sie hat. Die Hashcash-Verifikation bei 16 Bits dauert 70,2 ns, die Verifikation eines Nachweises der Tiefe 13 mit 16 Öffnungen 22.272,4 ns. Der Beweiser für den Browser ist ein WebAssembly-Modul von 14.744 Bytes. In der End-to-End-Prüfung im Browser löste Headless Chromium die Challenge der Zwischenseite mit WebAssembly in 29 bis 34 ms und mit dem JavaScript-Fallback in 84 bis 91 ms[16]. Die Primitive-Suite misst nur die Verifikation; eine native Zeitmessung steht nicht für die Lösungszeit in einem Browser.

Sitzungen verwenden ein symmetrisches Token. Der Verifizierer ist der Aussteller oder ein Cluster, der einen gemeinsamen Seed teilt; eine Pseudozufallsfunktion mit Schlüssel liefert also genau die nötige Authentifizierung. Ein Token mit BLAKE3 mit Schlüssel verifiziert sich in 155,8 ns, ein Ed25519-Token in 40.587,5 ns; eine Signaturprüfung kostet also etwa so viel wie 260 MAC-Prüfungen. Für die Wahl gibt es einen zweiten Grund: Shors Algorithmus bricht die diskreten Logarithmen, auf denen Ed25519 beruht, während eine Quantensuche gegen einen Hash mit Schlüssel höchstens eine quadratische Beschleunigung bringt. Ed25519 bleibt eine Option für Installationen, die Verifizierer brauchen, die selbst keine Tokens ausstellen können.

[15] Handbuch, Teil IV, „Der Challenge-Rundlauf“ und „Das Sitzungscookie“; SID 0010, „The browser solver“.

[16] `benchmarks/results/primitives-latest.md`; die WebAssembly-Größe aus `primitives-latest.json`, wie in Teil VIII des Handbuchs, „Speicherbedarf“, angegeben; die Lösungszeiten im Browser aus SID 0010, Kriterium 3 (Chromium 155, `cargo xtask browser`); die Erkennung der SHA-Instruktionen zur Laufzeit aus SID 0006, Revisionsnotiz (Teile A bis C).

6 Gemeinsamer Zustand ohne eigenes Konsensprotokoll

Lamport über verteilte Invarianten: Sicherheit, Lebendigkeit und wem das Log gehört

1987 beschrieb Leslie Lamport ein verteiltes System als eines, in dem der Ausfall eines Computers, von dessen Existenz man nicht einmal wusste, den eigenen Computer unbrauchbar machen kann. Als Rat gelesen, verlangt das von uns, die Computer zu zählen, von denen wir abhängen, und anzugeben, was gelten muss, wenn einer von ihnen ausfällt. Zwei Arten von Aussagen genügen. *Sicherheit* (Safety): Nie geschieht etwas Schlechtes, zum Beispiel handelt ein Knoten nie nach einer Richtlinie, die keine festgeschriebene Transaktion geschrieben hat. *Lebendigkeit* (Liveness): Irgendwann geschieht etwas Gutes, zum Beispiel bedient ein Knoten, der die Datenbank erreicht, irgendwann die neueste Richtlinie. Ein Konsensprotokoll ist ein eleganter Weg, beides zu erreichen. Es ist aber auch ein großes Stück Software, das man besitzen muss, und die Frage für eine Firewall ist nicht, ob Konsens gut ist, sondern wer das Log pflegen soll.

6.1 Warum SQL und warum kein eigenes Konsensprotokoll

Sibuna speichert wenig: einige hundert Richtlinienzeilen, eine Reputationstabelle und forensische Vorfälle, die Betreiber auch in andere Systeme exportieren. Dieser Zustand braucht keine maßgeschneiderte replizierte Engine. Er braucht einen zuverlässigen SQL-Speicher, lokal für einen Knoten und gemeinsam für viele. Ein einzelner Knoten hält seinen Zustand daher in einer eingebetteten SQLite-Datei in seinem Datenverzeichnis, im WAL-Modus, und ein Cluster richtet jeden Knoten auf eine gemeinsame PostgreSQL-Datenbank aus[17].

Wir haben uns dagegen entschieden, in der Firewall ein Konsensprotokoll zu betreiben. Eine Konsens-Engine ist ein großes System, das man neben einer Firewall besitzen muss, mit eigenen Formaten und eigenen Fehlerarten, und sie macht das Herunterfahren und jedes Upgrade aufwendiger. Verfügbarkeit und Replikation des Cluster-Zustands sind stattdessen Sache von PostgreSQL, bedient durch dessen Streaming-Replikation und Failover-Werkzeuge oder durch einen verwalteten Dienst. Wir halten das für die bescheidenere Wahl: Die Replikation von PostgreSQL hat sich in weit mehr Installationen bewährt, als unsere es je könnte, und Sibunas Schreibrate (gebündelte Vorfälle und seltene Richtlinienänderungen) ist so klein, dass ein einziger Primary kein Engpass ist.

Auch die Speicherung bleibt optional. Ohne `--data-dir` oder `--database-url` ist der Daemon eine Firewall im Arbeitsspeicher, deren Richtlinien aus einer Datei stammen.

6.2 Ein einziger Speicherthread

Ein einziger Speicherthread besitzt die Datenbankverbindungen; nichts anderes berührt SQL. Anfragethreads warten nie auf ihn. Sie schieben Vorfallsdatensätze in einen begrenzten Ring mit 512 Slots, der einen Datensatz verwirft und zählt, wenn er voll ist, und sie lesen die veröffentlichte Engine. Standardmäßig alle 500 ms führt der Speicherthread einen Takt aus: Er schreibt bis zu 32 wartende Vorfälle in einer Transaktion, liest die Richtlinienrevision und baut, wenn sie sich geändert hat, die Engine neu und veröffentlicht sie. Ein Neubau liest die Revision vor und nach dem Laden; hat ein Schreiber dazwischen eine Transaktion festgeschrieben, wird der Kandidat verworfen, und die alte Engine bleibt in Betrieb. Betreiber dürfen die Richtlinientabellen direkt beschreiben: Trigger erhöhen die Revision in der Transaktion des Schreibers, und unter PostgreSQL lässt ein `NOTIFY` die Knoten sofort einen Takt ausführen, während das regelmäßige Abfragen die Garantie bleibt[18].

Jeder Knoten eines Clusters hat eine eigene `--cluster-node`-Kennung zwischen 1 und 8.388.607. Challenge-Schlüssel sind an den ausstellenden Knoten gebunden; eine Lösung muss also zu dem Knoten zurückkehren, der ihre Challenge ausgestellt hat, während Sitzungstokens und Anforderungstickets auf jedem Knoten gültig sind, der den Seed teilt. Vorfallskennungen tragen den Knoten in ihren oberen Bits, und jeder Knoten führt eine dauerhafte Quittung, die in derselben Transaktion wie sein Batch fortgeschrieben wird; ein Batch, der nach einem mehrdeutigen Fehler wiederholt wird, schreibt also nichts doppelt.

Invarianten der Speicherschicht (SID 0011). **S1 (ein Besitzer).** Alles SQL läuft auf dem Speicherthread; kein Anfragethread wartet je auf die Datenbank. **S2 (vollständige Engines).** Jede Engine, die eine Anfrage liest, wurde vollständig gebaut, bevor sie veröffentlicht wurde, aus einer einzigen konsistenten Revision der Richtlinientabellen. **L1 (schließliche Veröffentlichung).** Eine festgeschriebene Änderung an den Richtlinientabellen erreicht jeden Knoten, der die Datenbank erreicht, bis zu seinem nächsten Takt. **F1 (geordneter Verlust).** Ist die Datenbank nicht erreichbar, bedient die zuletzt veröffentlichte Engine weiter, Vorfälle stauen sich im begrenzten Ring und werden mit einem Zähler verworfen, sobald er voll ist, und der Speicherthread versucht es mit Backoff erneut.

[17] SID 0011, „Abstract“, „Motivation and scope“ und „Why PostgreSQL for clusters“; SID 0005 beschreibt den früheren replizierten Entwurf.
[18] SID 0011, „The storage thread“, „Policy revision“ und „Incidents and campaigns“.

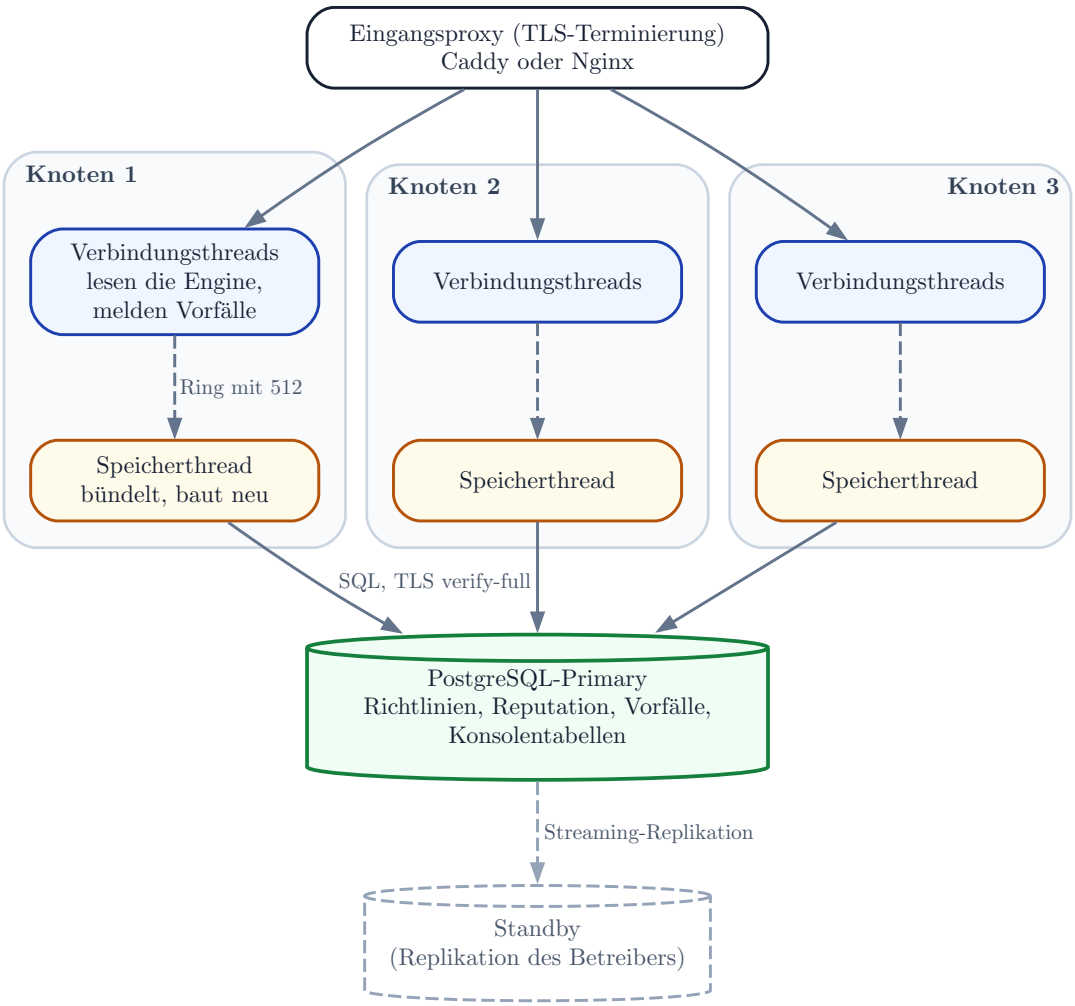


Abb. 4 Ein Sibuna-Cluster. Jeder Knoten hält seinen Anfragepfad im Speicher und überlässt alles SQL einem einzigen Speicherthread; die Knoten teilen sich eine PostgreSQL-Datenbank, deren Replikation und Failover Sache von PostgreSQL sind (SID 0011). Ein einzelner Knoten verwendet stattdessen eine eingebettete SQLite-Datei.

6.3 Ein typisiertes Schema

Das Schema drückt in SQL aus, was die Typen des Programms ausdrücken; die Datenbank lehnt also jede Zeile ab, die die Speicherschicht nicht darstellen könnte. SQLite-Tabellen sind `STRICT`; eine Aufzählungsspalte hat einen `CHECK`, der genau die Namen aufführt, die das Programm schreibt; eine begrenzte Ganzzahl hat eine Bereichsprüfung; und eine Spalte ist genau dann `NOT NULL`, wenn der Wert nie fehlt. Jede Anweisung bindet ihre Werte als Parameter, kein SQL entsteht also durch Quoting. Der Speicher der Konsole folgt denselben Regeln: eine Basis-Migration mit 33 typisierten Tabellen und ohne Trigger. Jede Invariante der Konsole, etwa „jede Änderung an einem Token schreibt eine Audit-Zeile“, setzt genau eine Funktion durch, die ihre Anweisungen in einer Transaktion ausführt, auf SQLite wie auf PostgreSQL[19].

Verbindungen zu PostgreSQL werden so geschützt, wie es die URL angibt, mit `sslmode` und `sslrootcert` von libpq, wenn der Daemon mit dem Feature `postgres-tls` gebaut ist. Ein Build ohne dieses Feature lehnt `require`, `verify-ca` und `verify-full` beim Start ab; er weicht nie auf Klartext aus. Produktive Cluster verwenden `sslmode=verify-full`.

Ähnliche Vorfälle werden zu Kampagnen gruppiert. Jeder Vorfall trägt ein Embedding mit 64 Dimensionen, gespeichert als 256 Bytes. Der Speicherthread hält die jüngsten 16.384 Embeddings aller Knoten im Speicher (4 MiB) und ordnet einen Vorfall der Kampagne seines nächsten Nachbarn zu, wenn der Cosinus-Abstand höchstens 0,35 beträgt. Dass gegen ältere Vorfälle nicht gruppiert wird, ist eine bewusste Auslassung: Kampagnen sind ihrer Natur nach jung, und die Schranke hält Speicher und Arbeit fest.

6.4 Was dieser Entwurf nicht leistet

Wir sollten die Grenzen dieser Wahl klar benennen. Sibuna enthält kein eigenes Konsensprotokoll, und die Konsistenz eines Clusters ist Sache von PostgreSQL: Ein Knoten, der die Datenbank verliert, behält seine letzte Richtlinie, bis er sich wieder verbindet. Eine auf einem Knoten ausgestellte Sitzung akzeptiert jeder Knoten, und eine auf einem Knoten verbrauchte Lösung wird abgewiesen, wenn sie einem anderen erneut vorgelegt wird, weil die Knoten einen Seed

[19]SID 0011, „Schema and migrations“ und sein Review-Verlauf (Transportsicherheit); SID 0013, „Abstract“.

teilen und Challenge-Schlüssel an den ausstellenden Knoten gebunden sind; keine der beiden Eigenschaften braucht die Datenbank. SID 0011 ist festgeschrieben: Ein Funktionstest mit drei Knoten auf einer PostgreSQL-Datenbank zeigt, dass eine auf einem Knoten eingefügte Richtlinie innerhalb eines Takts von allen dreien durchgesetzt wird, dass eine Honeypot-Sperre auf einem Knoten die anderen erreicht und dass Vorfallskennungen nie kollidieren. Noch treibt kein Messprogramm mehrere Knoten auf einer PostgreSQL-Datenbank unter Last; dieses Papier berichtet daher keine Cluster-Messungen. Die ausführbaren Dateien der Releases sind ohne PostgreSQL-Unterstützung gebaut; für einen Cluster baut man Sibuna aus dem Quellcode mit dem Feature `postgres-tls`[20].

7 Die Konsole

Sibuna enthält eine optionale Verwaltungskonsole: authentifizierte Dashboards mit Live-Datenverkehr, einen Untersuchungspfad von einer Ablehnung zu ihren aufgezeichneten Belegen, Bearbeitung von Richtlinien, Adressgruppen, Anreicherung mit Länderdaten aus offenen GeoIP-Daten, Benutzer mit Argon2id-Passwörtern und TOTP als zweitem Faktor, einen Audit-Trail, Webhooks und die Verwaltung eines Clusters. Sie ist in dieselbe Binärdatei kompiliert, und ihre Oberfläche läuft im Browser als WebAssembly-Modul[21].

Die bestimmende Bedingung der Konsole ist ihre Isolation vom Anfragepfad. Sie darf die Zähler der Datenebene und ihre Datenbank lesen und die Datenbank beschreiben, aber sie betritt nie einen Anfragethread, allokiert nie auf einem und hält nie eine Sperre, die ein Verbindungsthread braucht. Ihr Listener lässt höchstens 96 Verbindungen zu, von denen 16 für HTTP reserviert sind; 64 dürfen Browser-Streams und 16 Peer-Streams sein. In 1.0.0-beta.3 misst das vollständige Oberflächenmodul 724.330 Bytes (272.568 Bytes mit gzip komprimiert). Das liegt unter der Obergrenze von 768 KiB, die der Build durchsetzt, und über der Prüfschwelle von 640 KiB, was SID 0007 zur Überprüfung festhält[22].

Die Konsole baut ausgehende TLS-Verbindungen für langlebige Peer-Streams, Webhooks und GeoIP-Downloads auf. Zertifikate werden gegen die System-Wurzelzertifikate oder eine CA-Datei des Betreibers verifiziert, die Prüfung des Hostnamens lässt sich nicht abschalten, einfaches HTTP ist nur zu literalen Loopback-Adressen erlaubt, und ein Zertifikatsfehler fällt nie auf Klartext zurück. Eingehendes TLS terminiert Sibuna weiterhin nicht; das bleibt die Aufgabe des Eingangsproxys.

SID 0007 legt das Leistungsziel der Konsole fest: Ihre Aktivierung darf höchstens 1 % des Durchsatzes der Datenebene und höchstens 10 % der p99-Latenz kosten. Es gibt eine Messung: Mit einkompilierter, aber deaktivierter Konsole, nur mit Speicherung oder aktiviert und im Leerlauf, blieben die 16 Arbeitslasten des Gesamtprodukts innerhalb von

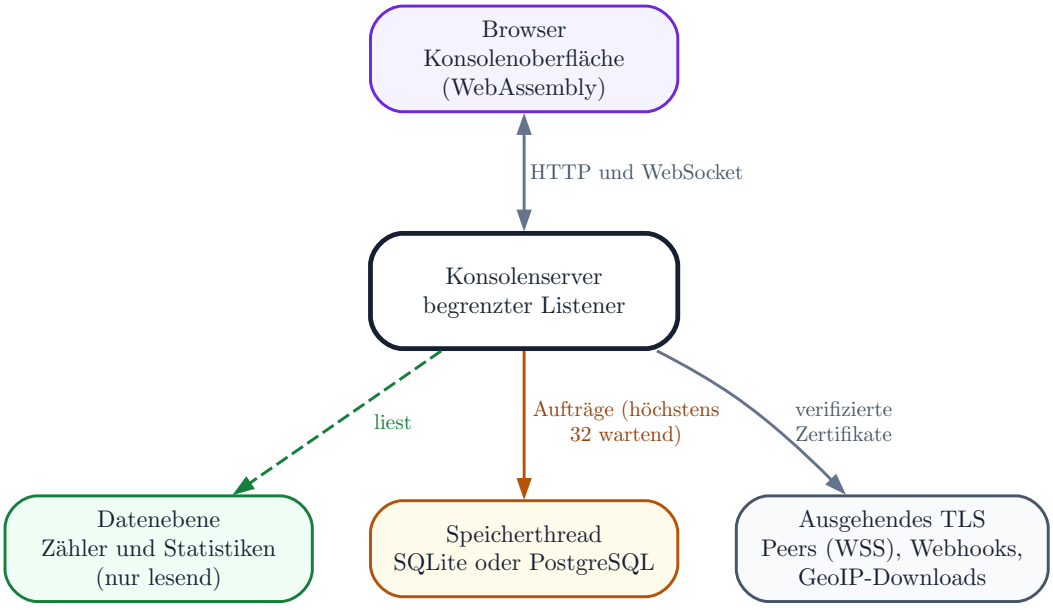


Abb. 5 Die Konsole. Der Browser führt die Oberfläche als WebAssembly-Modul aus; der Konsolenserver liest die Zähler der Datenebene und überlässt sein SQL dem Speicherthread; ausgehende Verbindungen zu Peers, Webhooks und GeoIP-Anbietern verwenden TLS mit verifizierten Zertifikaten (SIDs 0007, 0012, 0013).

[20] SID 0011, „Security and operational considerations“ und „Acceptance criteria and implementation status“ (Kriterium 2, `crates/storage/tests/multi_node.rs`); `.github/release-notes.txt`, „Known limitations“; SID 0002, „Authentication and work“ (an den Knoten gebundene Challenge-Schlüssel); Handbuch, Teil VIII, „Nicht gemessen“.

[21] SID 0007, „Abstract“ und seine Revisionsnotizen (Zulassung am Listener, Oberflächenmodul, GeoIP und Passwörter); SID 0010, Review-Verlauf (Portierung der Konsole, TOTP); SID 0012; SID 0013. Die Isolationsmessung steht im Review-Verlauf von SID 0010 (Portierung der Konsole); Teil VIII des Handbuchs, „Overhead der Konsole“, merkt an, dass das Dokument dafür keine Ergebnisdatei nennt.

[22] SID 0007, Review-Verlauf (Rust-Portierung, Konsolidierung; Integration in den Daemon), der 719.406 Bytes vor den JA4-Feldern von SID 0014 verzeichnet. Die Größe für 1.0.0-beta.3 gilt für das Modul, so gebaut, wie das Build-Skript des Daemons es baut (`cargo rustc --lib -p sibuna-console-ui --target wasm32-unknown-unknown --profile wasm -- -C link-arg=--compress-relocations`), komprimiert mit `gzip -9`.

Tabelle 4 Acht Traffic-Dashboards auf einem SQLite-Knoten mit seiner Konsole, in Headless Chromium 155 (`benchmarks/results/console-8-dashboards-2026-10-15.md`, aufgezeichnet am 2026-10-15). Ergebnis: PASS

Messgröße	Wert
Anfragen der Datenebene	6.000 in 30,0 s (200 pro Sekunde, eine Verbindung pro Anfrage)
Beantwortet	6.000: 200 x 1.200, 401 x 3.000, 403 x 600, 404 x 600, 500 x 600
Fehler (keine Antwort)	0
Latenz p50 / p90 / p99 / max	246 / 311 / 373 / 552 µs (einschließlich Verbindungsaufbau)
Residenter Speicher des Daemons beim Start, mit 8 Dashboards, nach dem Datenverkehr	36.152 KiB; 38.028 KiB; 41.496 KiB
Dashboards	3 Administratorsitzungen und 5 Wandanzeigen; jede erhielt 134 bis 139 `stats`-Deltas und endete mit denselben Summen, ohne Ausnahme, Konsolenfehler oder Verstoß gegen die Content-Security-Policy

etwa 1 % eines ohne Konsole gebauten Daemons, also im Rauschen. Das formale Abnahmekriterium, mit acht aktiven Dashboards unter Last, wurde nicht ausgeführt; wir behaupten daher nicht, dass das Ziel erreicht ist. Ein Funktionslauf mit acht Live-Dashboards wurde bestanden.

8 Empirische Bewertung

8.1 Methode

Drei Messprogramme, ausgeführt über `cargo xtask bench`, messen Sibuna. Die *Primitive*-Suite misst die Operationen des Anfragepfads in sieben gemessenen Batches nach einem ungemessenen Aufwärmen, baut veränderlichen Zustand außerhalb der Zeitmessung neu auf und meldet je Operation den Median-Batch, daneben den schnellsten und den langsamsten Batch. Das Messprogramm für das *Gesamtprodukt* treibt den Daemon mit `wrk` (2 Threads, 64 Verbindungen, 5 Sekunden, 3 Wiederholungen) im Forward-Auth- und im Reverse-Proxy-Modus, auf den Betriebsformen Gate und Shield, über vier Arbeitslasten: `admitted` (eine gültige Sitzung), `challenged` (keine Sitzung), `allowed_static` (`/robots.txt`) und `attack` (eine Sitzung plus `q=' OR 1=1--`, die nur Shield abweist). Das *Latenz*-Messprogramm nutzt einen Generator mit offener Schleife, der mit fester Rate sendet, gleich was der Daemon tut, und jede Anfrage ab ihrem Fälligkeitszeitpunkt misst; Warteschlangenzeit im Daemon wird so dem Daemon angerechnet[23].

Die Läufe wurden am 15. Oktober 2026 auf einem einzigen Host aufgezeichnet: einem AMD Ryzen 9 5950X (16 Kerne, 32 Threads) unter Linux 7.0.0-28. Der Daemon lief auf den CPUs 0 bis 3 mit vier Workern, der Lastgenerator und ein Stub-Ursprungsserver auf den CPUs 8 bis 15 (dem anderen CCD) und die Primitive-Suite auf CPU 4. Der Daemon war der Standard-Release-Build mit einkompiliertem SQLite und einkompilierter Konsole und lief im Speicher, ohne Speicherung.

Messumfang. Jeder Lauf nutzt einen Host über Loopback: kein TLS, kein Weitverkehrsnetz, keine echten Browser. Die Messprogramme zählen keine Allokationen; den allokationsfreien Anfragepfad prüfen stattdessen die Tests im eingeschwungenen Zustand. Sie führen die Release-Binärdatei unverändert aus, identifiziert über ihren SHA-256. Eine Rangfolge auf diesem Host ist keine Rangfolge auf einem anderen.

8.2 Latenzen der Primitiven

Tabelle 5 Zeitmessungen der Primitiven von Sibuna in Nanosekunden pro Operation auf einem Kern des Ryzen 9 5950X: der Median von sieben Batches, mit dem schnellsten und dem langsamsten Batch (`benchmarks/results/primitives-latest.md`, erzeugt 2026-10-15T22:34:59Z)

Subsystem	Arbeitslast	Median	Min.-Max.
Proof of Work	Hashcash-Verifikation, 16 Bits	70,2	69,9-71,0
Proof of Work	PoSW-Verifikation, Tiefe 13, 16 Öffnungen	22.272,4	22.257,0-22.316,4
Bot-Erkennung	Aho-Corasick, 40 Signaturen	55,7	55,3-57,2
Bot-Erkennung	Sequenzielle Teilzeichenkettensuche, 40 Signaturen (zum Vergleich)	763,0	762,2-763,5
IP-Klassifikator	IPv4-CIDR-Suche	32,0	31,8-32,2
IP-Klassifikator	IPv6-CIDR-Suche	79,3	78,5-79,6
Token	Sitzungstoken mit BLAKE3 mit Schlüssel	155,8	155,3-156,4
Token	Ed25519-Sitzungstoken	40.587,5	40.544,6-40.631,1

[23] `benchmarks/results/README.md`, „Method“ und „Provenance and limits“ (rustc 1.99.0; der Daemon ist `cargo build --release -p sibuna`, 49.113.856 Bytes); Handbuch, Teil VIII, „Methodik“.

Subsystem	Arbeitslast	Median	Min.-Max.
Verbrauchsmenge	Robin Hood, verbrauchen und nachschlagen	21,0	20,8-21,3
Ratenbegrenzer	GCRA, gehashter Client	6,2	6,1-6,3
Ratenbegrenzer	GCRA, vier Regelbereiche, gehashter Client	6,3	6,2-6,4
Ratenbegrenzer	GCRA, Client als Bytes (inklusive Hash mit Schlüssel)	12,8	12,7-12,8
Ratenbegrenzer	GCRA, vier Regelbereiche, Client als Bytes	13,1	13,0-13,2
HTTP-Parser	Zero-Copy-Anfrage und Cookie	132,5	132,3-133,2
Richtlinien-Engine	Vollständige Klassifikation, Shield-Profil	790,8	787,2-793,1
Richtlinien-Engine	Klassifikation, Gate-Profil	75,9	75,6-76,1
WAF	8-KiB-Body, semantischer Scan	12.784,3	12.771,5-12.809,2
WAF	8-KiB-Body, SQL-Injection im Audit-Modus	15.485,4	15.460,9-15.530,1

Die Streuungen sind gering: Kein Batch einer Zeile liegt mehr als 4,2 % vom Median seiner Zeile entfernt[24]. Der Daemon nutzt die Zeilen mit gehashtem Client, weil er den Client einmal pro Anfrage hasht; die Zeilen, die rohe Bytes übergeben, enthalten diesen Hash mit Schlüssel bei jeder Prüfung. Die Zeitmessung umschließt ganze Batches, nie einzelne Operationen; die Tabelle nennt daher keine Perzentile pro Operation.

8.3 Das Gesamtprodukt

Tabelle 6 Sibuna unter `wrk`: Anfragen pro Sekunde, Latenz in Mikrosekunden, CPU-Zeit pro Anfrage in Mikrosekunden und maximaler residenter Speicher; Mediane von drei Läufen (`benchmarks/results/product-latest.md`, erzeugt 2026-10-15T23:15:24Z)

Modus	Betriebs-form	Arbeitslast (Status)	Anfr./s	p50	p99	CPU	Max. KiB
Forward-Auth	Gate	`admitted` (200)	211.345	151	305	11,84	15.912
Forward-Auth	Gate	`challenged` (401)	198.315	160	326	13,15	16.176
Forward-Auth	Gate	`allowed_static` (200)	211.053	152	305	11,73	16.176
Forward-Auth	Gate	`attack` (200)	211.618	151	305	11,84	16.176
Forward-Auth	Shield	`admitted` (200)	208.101	152	308	12,87	15.928
Forward-Auth	Shield	`challenged` (401)	197.532	161	327	13,77	16.192
Forward-Auth	Shield	`allowed_static` (200)	211.636	151	305	11,78	16.192
Forward-Auth	Shield	`attack` (403)	210.549	152	307	12,32	16.192
Reverse Proxy	Gate	`admitted` (200)	142.498	427	768	27,68	15.896
Reverse Proxy	Gate	`challenged` (200)	199.465	160	324	12,98	16.160
Reverse Proxy	Gate	`allowed_static` (200)	147.303	411	759	26,88	16.324
Reverse Proxy	Gate	`attack` (200)	143.900	422	745	27,67	16.348
Reverse Proxy	Shield	`admitted` (200)	138.965	438	857	28,57	16.064
Reverse Proxy	Shield	`challenged` (200)	195.188	162	328	13,73	16.328
Reverse Proxy	Shield	`allowed_static` (200)	145.673	415	746	27,27	16.328
Reverse Proxy	Shield	`attack` (403)	209.507	151	306	12,17	16.328

Im Forward-Auth-Modus fragt der Eingangsproxy Sibuna, ob eine Anfrage zugelassen werden soll, und es gibt keinen Ursprungsserver; jede Arbeitslast liegt zwischen 197.532 und 211.636 Anfragen pro Sekunde, mit einem p99 von 305 bis 327 µs. Die vier CPUs des Daemons sind dort nicht ausgelastet (zwischen 2,47 und 2,72 von ihnen sind belegt), was auf den Lastgenerator als Obergrenze hindeutet. Im Reverse-Proxy-Modus enthalten die Zeilen der zugelassenen und der statischen Anfragen den Stub-Ursprungsserver und einen weiteren Loopback-Hop, und die CPUs des Daemons sind ausgelastet (3,95 bis 3,98 von vier belegt); Anfragen, die Sibuna selbst beantwortet (Challenges und die Abweisung des Angriffs durch Shield), laufen weiterhin mit etwa 195.000 bis 210.000 pro Sekunde. Der maximale residente Speicher bleibt in jeder Arbeitslast zwischen 15.896 und 16.348 KiB[25].

Einige wenige von rund zwei Millionen Antworten in drei Reverse-Proxy-Zeilen waren `503`: `wrk` verband sich neu, bevor eine schließende Verbindung ihren Platz unter dem Verbindungslimit pro Adresse zurückgegeben hatte. Der Daemon gibt den Platz jetzt zurück, bevor er die schließende Antwort schreibt; diese Dateien stammen aus der Zeit vor dieser Korrektur.

[24] Handbuch, Teil VIII, „Latenzen der Primitiven“, berechnet aus `primitives-latest.json`.

[25] `benchmarks/results/product-latest.md`; belegte CPUs aus `cores_busy_median` in `product-latest.json`; die `503`-Antworten und ihre Korrektur aus `benchmarks/results/README.md` und SID 0002, Revisionsnotiz (Fehlerbehebung).

Tabelle 7 Ressourcenbedarf des Sibuna-Daemons (`product-latest.json` und `primitives-latest.json`, 15. Oktober 2026, wie in Teil VIII des Handbuchs tabelliert)

Messgröße	Wert
Binärdatei des Daemons, Standard-Build mit SQLite und Konsole	49.113.856 Bytes
Residenter Speicher im Leerlauf, Produktläufe	5.592 bis 5.788 KiB
Maximaler residenter Speicher während einer Arbeitslast	15.896 bis 16.348 KiB
Residenter Speicher nach den Arbeitslasten	16.176 bis 16.348 KiB
Tabellen der Richtlinien-Engine	209.052 Bytes
Tabelle der Bot-Signaturen	42.244 Bytes
Tabelle der WAF-Signaturen	164.908 Bytes
Beweiser für den Browser (WebAssembly)	14.744 Bytes

Eine spätere Änderung am Release-Profil verwirft die Debug-Informationen des mitgelieferten SQLite, laut der Notiz im Build-Manifest etwa 13 MB der 49 MB; die Ergebnisdateien wurden seitdem nicht neu erzeugt, daher nennen wir die gemessene Größe[26].

8.4 Latenz bei gleicher angebotener Last

`wrk` hält 64 Anfragen gleichzeitig in Bearbeitung; die Anfragerate hängt also vom Daemon ab, und die Perzentile von `wrk` lassen die Anfragen aus, die es nicht gesendet hat, während es wartete. Das Messprogramm mit offener Schleife legt stattdessen die angebotene Rate fest und rechnet jede Verzögerung dem Daemon an; genau das erlebt ein Besucher.

Im Forward-Auth-Modus bleibt das p99.9 bis 150.000 Anfragen pro Sekunde unter 75 µs. Im Reverse-Proxy-Modus, mit dem Ursprungsserver im Pfad, liegen bei 100.000 Anfragen pro Sekunde das p99 bei 100 µs und das p99.9 bei 150 µs. Bei 150.000 Anfragen pro Sekunde im Reverse-Proxy-Modus ist der Daemon über seiner Kapazität: Er schafft 134.344 Anfragen pro Sekunde, nahe an den 138.965, die `wrk` für dieselbe Arbeitslast gemessen hat, und die Latenz wächst auf Sekunden, wie es ein Test mit offener Schleife oberhalb der Kapazität zeigen sollte. Die vier Fehler dieser Zeile traten in der Überlast auf[27].

8.5 Was die Messung verändert hat

Das Latenz-Messprogramm hat den Entwurf der Verbindungsthreads verändert. In seinen ersten Läufen, mit 64 Verbindungen, die der Daemon jeweils nach 4.096 Anfragen schloss, lag das 99,9. Perzentil bei 100.000 zugelassenen Anfragen pro Sekunde bei 0,3 bis 2 ms, ein Verteilungsende, das `wrk` nicht gezeigt hatte. Einen Thread zu erzeugen und zu beenden bildet seinen Signal-Stack ab und wieder aus und gibt seine Stack-Seiten zurück, und diese Änderungen am Adressraum ließen die anderen Threads stocken. Ein Experiment klärte die Diagnose: Ein Build, der nie Verbindungen schloss und daher während des Laufs keinen Thread erzeugte, ergab bei derselben Rate 54 bis 60 µs. Die Korrektur lässt einen Thread, der seine Verbindung beendet hat, parken und die nächste übernehmen; ein Thread, der 10 Sekunden lang geparkt war, endet, damit ein Burst seine Stacks nicht resident zurücklässt. Nach der Korrektur liegt das p99.9 bei 100.000 Anfragen pro Sekunde im Forward-Auth-Modus bei 62 µs und im Reverse-Proxy-Modus bei 150 µs[28].

Ein seltenes, periodisches Stocken verschiebt das äußerste Ende der Verteilung lange bevor es das p99 verschiebt. Bei 100.000 Anfragen pro Sekunde über 64 Verbindungen dauert jede Verbindung $4096/1562.5 \approx 2.6$ s; jede Sekunde

Tabelle 8 Sibuna, Shield, zugelassene Anfragen bei festen angebotenen Raten; Latenz in Mikrosekunden und CPU-Zeit pro Anfrage, Mediane von drei Läufen zu je 10 Sekunden (`benchmarks/results/latency-latest.md`, erzeugt 2026-10-15T23:07:14Z)

Modus	Angebotene Anfr./s	Erreichte Anfr./s	p50	p90	p99	p99.9	Fehler	CPU µs
Forward-Auth	10.000	10.000	30	32	37	58	0	16,92
Forward-Auth	25.000	25.000	28	31	34	45	0	15,83
Forward-Auth	50.000	50.000	25	27	30	40	0	13,83
Forward-Auth	100.000	99.998	24	26	41	62	0	13,48
Forward-Auth	150.000	149.996	25	36	51	74	0	14,16
Reverse Proxy	10.000	10.000	56	61	65	76	0	32,25
Reverse Proxy	25.000	25.000	51	54	58	72	0	28,63
Reverse Proxy	50.000	50.000	47	50	60	74	0	27,45
Reverse Proxy	100.000	99.996	55	75	100	150	0	28,42
Reverse Proxy	150.000	134.344	583.934	1.544.160	1.906.273	1.967.134	4	29,22

[26] Handbuch, Teil VIII, „Speicherbedarf“; die `Cargo.toml` im Wurzelverzeichnis, Profil `[profile.release.package.libsqlite3-sys]`.
[27] `benchmarks/results/latency-latest.md`; die Zeilen bei 25.000 Anfragen pro Sekunde und die Reverse-Proxy-Zeilen bei 150.000 stehen in dieser Datei.
[28] SID 0002, Revisionsnotiz (Leistung, wiederverwendete Verbindungsthreads); Handbuch, Teil VIII, „Was die Messung verändert hat“; `benchmarks/results/latency-latest.md`.

starten und enden also etwa 24 Threads. In einem Lauf mit einer Million Anfragen umfasst das oberste 0,1 % nur 1.000 Anfragen, und schon wenige verzögerte Anfragen pro Threadwechsel füllen es. Einen solchen Defekt sieht man nur mit dem p99.9 und einer Zeitmessung jeder einzelnen Anfrage.

8.6 Die wichtigsten Ergebnisse

1. *Klassifikation unter einer Mikrosekunde.* Im Gate-Profil klassifiziert die Richtlinien-Engine eine Browseranfrage in 75,9 ns; mit vollständiger Prüfung im Shield-Profil dauert es 790,8 ns.
2. *Symmetrische Verifikation.* Ein Sitzungstoken mit BLAKE3 mit Schlüssel verifiziert sich in 155,8 ns, ein Ed25519-Token in 40.587,5 ns: etwa 260-mal weniger Arbeit für dieselbe Integrität innerhalb einer Vertrauensdomäne.
3. *Ein kleiner, stabiler Ressourcenbedarf.* Rund 200.000 Forward-Auth-Anfragen pro Sekunde auf vier CPUs, mit einem maximalen residenten Speicher von etwa 16 MiB in jeder Arbeitslast und unter 6 MiB im Leerlauf.
4. *Verteilungsenden brauchen Messungen mit offener Schleife.* Das Stocken beim Threadwechsel blieb dem Messprogramm mit geschlossener Schleife verborgen und war beim p99.9 unter einem Generator mit offener Schleife offensichtlich.

9 Marktvergleich: Sibuna und seine Nachbarn

Wir vergleichen Sibuna mit den Werkzeugen, die ein Betreiber am ehesten gegen es abwägen wird. Die Angaben zu den anderen Produkten haben wir im September 2026 ihrer öffentlichen Dokumentation und ihren Release-Artefakten entnommen; das Sibuna-Handbuch verzeichnet sie mit ihren Quellen in Teil II[31]. Jedes dieser Produkte dient seinen Nutzern gut, und jedes hat Entwurfsentscheidungen getroffen, deren Vorteile unser Messprogramm nicht misst. Wir beschreiben die Unterschiede in der Architektur, keine Rangfolge.

Tabelle 9 Funktionsvergleich (Handbuch, Teil II; Angaben zu anderen Produkten im September 2026 der öffentlichen Dokumentation entnommen, die JA4-Zeile im Oktober 2026)

Funktion	Sibuna 1.0.0-beta.3	Anubis 1.27	SafeLine CE 9	Cloudflare WAF
Läuft als	Eine Binärdatei; Reverse Proxy oder Forward-Auth; optional eine SQLite-Datei oder eine gemeinsame PostgreSQL-Datenbank	Eine Go-Binärdatei; Reverse Proxy oder Forward-Auth	Sieben Docker-Container (tengine, detector, mgt, luigi, fvm, chaos, PostgreSQL)	Gehostetes Netzwerk; DNS zeigt auf Cloudflare
Laufzeit	Nativer Code, kein Garbage Collector; ohne Speicherung keine Abhängigkeit von Dritten	Go, mit Garbage Collection	Proxy und Detektor in C/C++, Verwaltung in Go, PostgreSQL	Proprietäre Edge-Software
Lizenz	LGPL-3.0-or-later	MIT	GPL-3.0-Verwaltungscode; geschlossene Detektor-Images	Proprietärer Dienst
Proof-of-Work-Zulassung	Hashcash (auf Bitebene) und sequenzielle Arbeit nach Cohen-Pietrzak, je Regel gewählt	SHA-256-Hashcash in Hex-Nibbles; Challenges ohne Arbeit	JavaScript-Challenge und CAPTCHA; nicht an Arbeit gebunden	Managed Challenge, JS Challenge, Turnstile
Challenge-Zustand auf dem Server	Keiner bis zur Lösung; gelöste Tags in einer festen Robin-Hood-Tabelle	Store-Backend: Speicher, bbolt, Valkey oder S3	Vom Stack verwaltet	Von Cloudflare verwaltet
Sitzungstoken	BLAKE3 mit Schlüssel, 75 Zeichen, an die bezahlte Arbeit gebunden; Ed25519 optional	JSON Web Token, mit Ed25519 signiert; HS512 optional	Vom Stack ausgestelltes Cookie	Cookie `cf_clearance`
Prüfung der Nutzlast	Shield: markierter Automat plus strukturelle Tokenizer, erste 8 KiB des Bodys	Keine	Semantische Analyse über OWASP-Kategorien	Verwaltete Regelsätze (ab Pro)
TLS-Fingerabdruck (JA4)	Regeln vergleichen einen JA4-Wert, den ein vertrauenswürdiger Eingangsproxy weiterleitet; nicht von Sibuna berechnet	Nicht dokumentiert; Richtlinien können sich auf den HTTP-Fingerabdruck JA4H beziehen, den Anubis berechnet	Bei der eigenen TLS-Terminierung berechnet; eigene Regeln und eine eingebaute Sperrliste	JA3 und JA4 in eigenen Regeln und in der Analyse (Enterprise mit Bot Management)
Gemeinsamer Zustand	Eine PostgreSQL-Datenbank für Richtlinien, Reputation und Forensik	Gemeinsamer Signaturschlüssel; gemeinsamer Valkey-Speicher	Ein Stack pro Host	Globales Anycast-Netz
TLS-Terminierung	Nein (Eingangsproxy)	Nein (Eingangsproxy)	Ja	Ja

[31] Handbuch, Teil II, „Funktionsvergleich“ und „Zwischenseiten und Edges“; die Lizenz von Sibuna ist die `license` des Workspace in `Cargo.toml`.

Tabelle 10 Sibuna (Shield) und Anubis 1.27.0 unter `wrk`: Anfragen pro Sekunde und p99-Latenz in Mikrosekunden, Mediane von drei fünfsekündigen Läufen (`benchmarks/results/comparison-latest.md`). Der Status ist 200, sofern nicht anders angegeben; Shield weist den Angriff mit 403 ab.

Modus	Arbeitslast	Sibuna Anfr./s	Sibuna p99	Anubis Anfr./s	Anubis p99
Forward-Auth	`admitted`	208.635	308	36.509	7.477
Forward-Auth	`challenged`	193.087	342	113.170	3.781
Forward-Auth	`allowed_static`	212.444	304	89.904	4.547
Forward-Auth	`attack`	211.347 (403)	306	36.853	7.283
Reverse Proxy	`admitted`	103.298	3.321	19.398	11.062
Reverse Proxy	`challenged`	197.327	328	28.842	9.546
Reverse Proxy	`allowed_static`	106.819	3.316	28.862	30.124
Reverse Proxy	`attack`	210.483 (403)	306	23.399	14.124

9.1 Gemessener Vergleich mit Anubis

Wir haben die offizielle Release-Binärdatei von Anubis v1.27.0 neben Sibuna mit dem Messprogramm aus Das Gesamtprodukt gemessen: dieselben vier Arbeitslasten, dieselben vier CPUs für jeden Thread jedes Produkts, dieselbe `wrk`-Last, angeglichene Hashcash-Arbeit (8 Nullbits für Sibuna, 2 hexadezimale Nullziffern für Anubis), eine Sitzung, die wie in einem Browser durch Lösen der Challenge des jeweiligen Produkts erworben wurde, und `caddy respond` als Ursprungsserver[29]. Sibuna läuft hier in der Betriebsform Shield und prüft also zusätzlich jede Nutzlast, was Anubis nicht tut.

Wie diese Zeilen zu lesen sind. Diese Zahlen beschreiben vier Arbeitslasten auf einer Maschine, nicht jede Installation, und mehrere Zeilen vergleichen nicht Gleiches mit Gleichem. Im Forward-Auth-Modus beantwortet Anubis `/robots.txt` und den Angriffspfad selbst; Anubis prüft keine Nutzlasten, Shields Abweisung leistet also mehr Arbeit; der Einstieg in eine Challenge sind bei Sibuna zwei Anfragen und bei Anubis eine Seite; und das symmetrische Token erklärt einen Großteil des Abstands bei der Sitzungsprüfung, denn Anubis' eigenes HS512-Verfahren ist 55 % schneller als sein Ed25519-Standard. Anubis ist ein weit verbreitetes und leistungsfähiges Werkzeug, und seine Maintainer haben aus guten Gründen andere Entscheidungen getroffen. Was die Messung zeigt, ist, dass ein kleiner, begrenzter Anfragepfad die Zulassung für den Verteidiger billig machen kann.

Anubis verbraucht im Forward-Auth-Modus 106 µs CPU pro zugelassener Anfrage, Sibuna 13; sein maximaler residenter Speicher lag im Forward-Auth-Modus bei etwa 32 MiB und wuchs im Reverse-Proxy-Modus auf etwa 600 MiB, während der von Sibuna durchweg bei etwa 16 MiB blieb. Hinter `caddy respond` begrenzt der Ursprungsserver Sibunas weitergeleitete Zeilen, denn er teilt sich die CPUs mit dem Lastgenerator: Mit dem leichteren Stub-Ursprungsserver aus Das Gesamtprodukt erreichen dieselben Zeilen etwa 139.000 Anfragen pro Sekunde.

9.2 Detaillierter Architekturvergleich

9.2.1 SafeLine (Chaitin Technology)

SafeLine überträgt die semantische Idee von libinjection in ein vollständiges Produkt mit Verwaltungskonsole, und sein Hersteller berichtet Erkennungsraten von 71,65 % (ausgewogener Modus) und 76,17 % (strenger Modus) bei Fehlalarmraten von 0,07 % und 0,22 %[32]. Das sind Messungen der Erkennungsqualität, die Sibuna für seine eigene Engine nicht veröffentlicht hat.

- *Einsatz.* Die Community Edition wird als sieben Docker-Container auf einem Linux-Host mit mindestens einem CPU-Kern, 1 GB Arbeitsspeicher und 5 GB Festplatte ausgeliefert. Sie terminiert TLS selbst, was Sibuna nicht tut.
- *Zustand.* Sie führt ihr Angriffsprotokoll in PostgreSQL innerhalb des Stacks, ein Stack pro Host.

Tabelle 11 Zulassungsoperationen pro Sekunde auf zwei CPUs, Median von sieben Batches (`benchmarks/results/admission-latest.md`). Anubis läuft je einmal mit jedem seiner Token-Verfahren.

Produkt (Token)	Sitzungsprüfung	Nicht authentifiziert	Einstieg	Nachweisprüfung	Residenter Speicher
Sibuna (BLAKE3 mit Schlüssel)	88.104	78.997	42.664	57.924	7,6 MiB
Anubis (Ed25519)	10.951	28.619	7.893	12.668	32,6 MiB
Anubis (HS512)	16.976	27.785	7.952	19.261	29,1 MiB

[29] `benchmarks/results/comparison-latest.md` und `admission-latest.md`, aufgezeichnet am 16. Oktober 2026 mit `cargo xtask bench product --anubis` und `cargo xtask bench admission --anubis`; Teil VIII des Handbuchs, „Sibuna und Anubis“, nennt die Methode und jeden Unterschied zwischen den Produkten.
[32] Handbuch, Teil VIII, „Nicht gemessen“: vom Hersteller berichtete Zahlen, geprüft im September 2026.

- *Wo Sibuna sich unterscheidet.* Sibuna ist eine einzige Binärdatei, die mit einer optionalen SQLite-Datei startet, und ein Cluster teilt sich eine PostgreSQL-Datenbank, die der Betreiber womöglich bereits betreibt. Sein residenter Speicher im Leerlauf lag in den Produktläufen bei 5.592 bis 5.788 KiB. SafeLine haben wir auf unserem Host, der keinen Docker-Daemon hatte, nicht betrieben; wir stellen daher keinen Leistungsvergleich an.

9.2.2 Anubis und Coraza

Die Proof-of-Work-Zwischenseite wurde 2025 verbreitet, als der Datenverkehr von Scrapern für das Training von Sprachmodellen den Betrieb gewöhnlicher Ursprungsserver teuer machte, und Anubis ist ihre bekannteste offene Implementierung. Anubis ist ein leistungsfähiger, weit verbreiteter Reverse Proxy in Go: Er liefert ein JavaScript-Rätsel aus, signiert ein JSON Web Token mit Ed25519 und hält den Challenge-Zustand in einem Speicher. Coraza implementiert die Regelsprache von ModSecurity (SecLang) und führt das OWASP Core Rule Set aus; das ist ein Weg, neben einem solchen Gate eine regelbasierte Prüfung hinzuzufügen.

- *Wohin die Arbeit geht.* Anubis und Sibuna Gate lassen beide den Client zahlen, bevor der Ursprungsserver irgendetwas tut. Sibuna fügt eine zweite, je Regel gewählte Stufe sequenzieller Arbeit für Routen hinzu, die ein stärkeres Gate verdienen, und seine Sitzungen erinnern sich, wie viel Arbeit bezahlt wurde.
- *Was der Server sich merkt.* Anubis speichert Challenges; Sibuna stellt einen selbstauthentifizierenden Datensatz aus und merkt sich nur gelöste Tags, in einer festen Tabelle (Satz 3). Sein Standard-Token ist symmetrisch; es ist billiger zu verifizieren und lässt einem Quantenangreifer nur eine quadratische Beschleunigung.
- *Prüfung.* Sibuna Shield führt seine eigene strukturelle Prüfung im selben Durchgang wie die Zulassung aus; ein Prozess und ein Parser erfüllen also beide Aufgaben. Das OWASP Core Rule Set führt es nicht aus. Wir haben die vorhandenen SecLang-Engines geprüft, darunter Bindungen an libmodsecurity und Coraza, und keine aufgenommen: Eine Prüf-Engine auf einem Edge-Server ist sicherheitskritischer Code, den wir lieber selbst besitzen und prüfen. Falls Sibuna Kompatibilität mit dem Core Rule Set hinzufügt, legt zuerst ein Entwurfsdokument die Teilmenge fest[33].
- *Messung.* Auf einem Host unter derselben Last beantwortete Sibuna im Forward-Auth-Modus mehr als fünfmal so viele zugelassene Anfragen, mit niedrigerem p99 und etwa einem Achtel der CPU-Zeit pro Anfrage (Gemessener Vergleich mit Anubis).

9.2.3 Cloud-Edge-WAFs (Cloudflare und andere)

Gehostete Edge-Netze verteilen Prüfung, Ratenbegrenzung und Bot-Bewertung auf viele Points of Presence mit einer globalen Steuerungsebene. Sie fangen volumetrische Angriffe ab, die kein einzelner Server abwehren kann, und brauchen keine Hardware des Betreibers. Sibunas Cluster-Einsatz übernimmt ihre Form, eine gemeinsame Richtlinienebene, die lokale unveränderliche Schnappschüsse speist, und bleibt dabei selbst gehostet.

- *Datenhoheit.* Eine gehostete WAF terminiert das TLS des Betreibers und sieht die Nutzlasten der Anfragen auf gemeinsam genutzter Infrastruktur, was mit Anforderungen an den Speicherort der Daten kollidieren kann.
- *Tarife und Kontingente.* Funktionen sind an Tarife gebunden: Die veröffentlichten Kontingente von Cloudflare erlauben 5, 20, 100 und 1.000 eigene Regeln sowie 1, 2, 5 und 100 Regeln zur Ratenbegrenzung in den Tarifen Free, Pro, Business und Enterprise.
- *Wo Sibuna sich unterscheidet.* Sibuna läuft auf den eigenen Maschinen des Betreibers ohne Gebühr pro Anfrage, und seine Richtlinienänderungen erreichen jeden Knoten beim nächsten Speichertakt (standardmäßig 500 ms). Es wehrt keine volumetrischen Netzwerkangriffe ab, bewertet Bots nicht mit einem trainierten Modell und terminiert kein TLS; ein Einsatz, der das braucht, braucht weiterhin einen Eingangsproxy oder ein Edge-Netz davor.

10 Implementierung: Kleine Crates und wenige Abhängigkeiten

Sibuna ist in Rust geschrieben, als Workspace kleiner Crates, deren Abhängigkeiten in eine Richtung zeigen: wiederverwendbare Bibliotheken, dann ein Kern aus Konfiguration und Diagnose, dann die Anwendungsbibliotheken, dann der Daemon und der Beweiser für den Browser. Drei technische Entscheidungen sind für einen Betreiber wichtig; wir halten sie kurz fest[34].

Wenige Abhängigkeiten. Wir behandeln jede Bibliothek von Dritten als Verbindlichkeit: Sie muss geprüft, aktualisiert und für vertrauenswürdig befunden werden. Die Datenebene verwendet keine. Hashing, Signaturen, Base64, JSON, DEFLATE, Aho-Corasick und der HTTP/1.1-Parser sind kleine eigene Bibliotheken, die andere Projekte auch eigenständig nutzen können. Eine Abhängigkeit wird nur aufgenommen, wenn sie ausgereift und in sich geschlossen ist und ein wirklich großes Problem löst, und die Begründung steht neben ihr im Build-Manifest und in einem Entwurfsdokument. Zwei wurden aufgenommen, jede auf eine Bibliothek beschränkt: `sqlx` für die Speicherung in SQLite und PostgreSQL, das etwa hundert transitive Crates mitbringt[35], und `rustls` für das ausgehende TLS der Konsole und das TLS zu PostgreSQL, dreizehn Crates zur Laufzeit, denn ein selbst geschriebener TLS-Client wäre ein sicherheitskritisches

[33] SID 0010, „The OWASP Core Rule Set“.
[34] SID 0010, „Workspace“, „Dependency policy“ und „Engineering standards“.
[35] SID 0011, „Why sqlx“; `Cargo.toml`, wo jedes Crate von Dritten seine Begründung trägt.

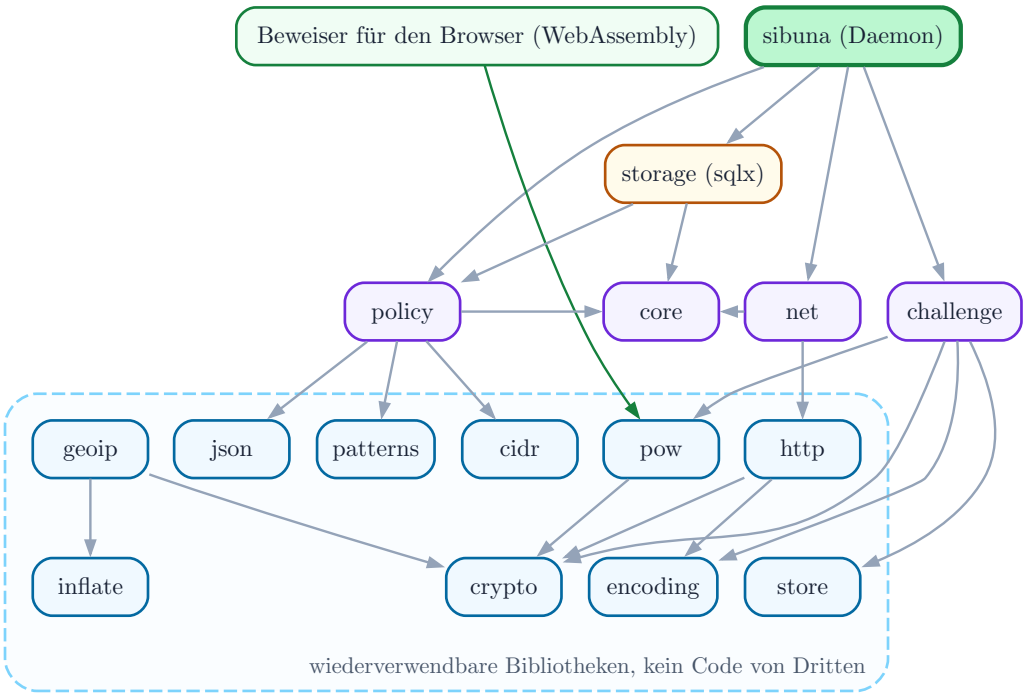


Abb. 6 Die Bibliotheken von Sibuna, vereinfacht (nach SID 0010). Pfeile zeigen von einer Bibliothek auf das, was sie nutzt. Wiederverwendbare Bibliotheken hängen von keiner Anwendungsbibliothek ab, und nur die Bibliotheken für Speicherung und Konsole bringen Code von Dritten mit.

Projekt, größer als Sibunas Datenebene[36]. Ein Build ohne Speicherung und Konsole hat überhaupt keine Abhängigkeit von Dritten.

Speichersicherheit. Code, der die Speichersicherheitsprüfungen des Compilers umgeht, ist standardmäßig verboten; ein Modul, das ihn braucht, muss namentlich zustimmen und jede Verwendung in einem Kommentar begründen. Einige Module tun das: die Pfade, die die SHA-Instruktionen des Prozessors nutzen, ein Schreibvorgang, der Geheimnisse löscht, die Deklarationen einiger Funktionen der C-Bibliothek für die Signalbehandlung, die Exporte der beiden WebAssembly-Module und der Ring, der Vorfälle zum Speicherthread trägt. Der Ring ist die einzige Stelle, an der wir solchen Code der Geschwindigkeit wegen behalten haben, und das erst nach einer Messung: Der begrenzte Kanal der Standardbibliothek kostete mit 16 Produzenten 20 bis 45 % mehr pro Push[37]. Andernorts übernimmt das Ownership-Modell die Arbeit, die sonst eine handgeschriebene Speicherverwaltung leisten müsste: Eine geparste Anfrage kann ihren Puffer nicht überleben, eine Engine wird mit ihrem letzten Leser freigegeben, und ein Socket kann nicht geschlossen werden, solange noch irgendein Thread ihn erreichen kann.

Disziplin und Tests. Ein Funktionsrumpf hat höchstens 70 logische Zeilen und eine Datei höchstens 1.408 Zeilen, durchgesetzt von einem Prüfwerkzeug. Bibliothekscode gibt strukturierte Fehler zurück und gibt nie etwas aus, und jeder Fehler, den ein Benutzer sieht, nennt seinen Ort, zeigt einen Ausschnitt und gibt einen Hinweis. Die Workspace-Testsuite von 1.0.0-beta.3 führt etwa 2.200 Tests auf SQLite und etwa 700 weitere gegen PostgreSQL aus; die Oberfläche der Konsole hat 298 Tests, darunter Golden Fixtures; und Browserprüfungen steuern Headless Chromium durch die Zwischenseite und durch jede Seite der Konsole[30].

11 Grenzen und offene Arbeit

Wir führen auf, was dieses Papier nicht belegt, damit keine Aussage weiter gelesen wird, als ihre Belege tragen.

- *Ein gemessener Nachbar.* Wir haben Anubis auf einem Host gemessen (Gemessener Vergleich mit Anubis). Weder SafeLine noch eine gehostete WAF haben wir gemessen.
- *Kein Cluster unter Last.* Ein Funktionstest betreibt drei Knoten auf einer PostgreSQL-Datenbank (SID 0011, festgeschrieben), aber kein Messprogramm treibt mehrere Knoten unter Last.

[36] SID 0012, „Specification“, „Rationale and alternatives“ und Review-Verlauf (PostgreSQL-TLS gebaut).
[37] SID 0010, Review-Verlauf (Sicherheitsprüfung, 2026-10-15): der begrenzte Standardkanal gegen den Vyukov-Ring, `tests/timing.rs`. Die Module mit Zustimmung sind `crates/crypto/src/sha256/` (x86-64 und AArch64), `crates/crypto/src/ct.rs`, `cmd/sibuna/src/signals.rs`, `cmd/sibuna-pow-wasm/src/lib.rs`, `cmd/console-ui/src/abi.rs` und `crates/store/src/ring.rs`; der Workspace setzt `unsafe_code = "deny"` und Clippys `undocumented_unsafe_blocks = "deny"`.
[30] SID 0010, „Acceptance criteria and implementation status“ (Kriterien 1, 3 und 5), das vor SID 0014 etwa 2.100 Workspace-Tests und 290 Tests der Oberfläche verzeichnete; die Zahlen für 1.0.0-beta.3 stammen aus `cargo test --workspace -- --list`.

- *PostgreSQL erfordert einen Build aus dem Quellcode.* Die ausführbaren Dateien der Releases haben SQLite-Speicherung und die Konsole, aber keine PostgreSQL-Unterstützung; für einen Cluster baut man Sibuna mit dem Feature `postgres-tls`.
- *Das Isolationskriterium der Konsole* mit acht aktiven Dashboards unter Last wurde nicht ausgeführt (Die Konsole).
- *Nur Loopback.* Keine Messung umfasst TLS, ein Weitverkehrsnetz oder echte Browser unter Last; ein Einsatz sollte seinen eigenen Pfad messen.
- *Begrenzte Prüfung.* Shield ist eine heuristische Prüfung der ersten 8 KiB eines Bodys, kein vollständiger SQL- oder HTML-Parser, und die Prüfung im Forward-Auth-Modus sieht nur die Bytes, die der Eingangsproxy sendet.
- *Protokollumfang.* Sibuna spricht HTTP/1.0 und HTTP/1.1; es hat kein HTTP/2 und keine Terminierung von eingehendem TLS.
- *TLS-Fingerabdrücke kommen vom Eingangsproxy.* Sibuna kann JA4 nicht selbst berechnen. Ein Fingerabdruck ist nur so vertrauenswürdig wie der Eingangsproxy, der ihn schreibt, und dieser Eingangsproxy muss jede Kopie überschreiben, die ein Client gesendet hat. Ein Client kann mit spezialisierten Bibliotheken den Handshake eines Browsers nachahmen; JA4 erhöht also die Kosten der Automatisierung, belegt aber keine Identität[11].
- *Kryptographische Gewissheit.* Die Stufe der sequenziellen Arbeit beruht auf veröffentlichten Beweisen im Random-Oracle-Modell; unser konkretes Protokoll hat Tests gegen eine Referenzimplementierung der Beschriftung, aber keine unabhängige kryptographische Prüfung.
- *Eine Maschine.* Jede Zahl wurde auf einem einzigen AMD Ryzen 9 5950X erhoben. Eine Rangfolge auf diesem Host ist keine Rangfolge auf einem anderen.

12 Fazit

Webabwehr ist oft durch Anlagerung gewachsen: interpretierte Laufzeitumgebungen, Container, Regelwerke aus regulären Ausdrücken und externe Speicher, vor einer Anwendung aufgetürmt. Mit Sibuna haben wir die entgegengesetzte Richtung versucht, geleitet von einigen schlichten Ideen:

- den Absender die Zulassung mit nachprüfbarer Arbeit bezahlen lassen, die Prüfung des Servers billig und zustandslos halten und eine Sitzung sich merken lassen, wie viel Arbeit sie bezahlt hat;
- nach Struktur in linearer Zeit prüfen, im selben Durchgang wie die Zulassung, sodass keine Sitzung die Prüfung umgehen kann;
- die Maschine respektieren: ein Puffer pro Verbindung, keine Allokation auf dem Anfragepfad, begrenzte Tabellen mit schlüsselbasierter Platzierung und Threads, die wiederverwendet statt neu erzeugt werden; und
- dauerhaften gemeinsamen Zustand einer ausgereiften Datenbank überlassen, hinter einem einzigen Speicherthread, auf den der Anfragepfad nie wartet.

Auf einer Maschine mit 16 Kernen erlauben diese Entscheidungen Sibuna, im Forward-Auth-Modus auf vier CPUs mehr als 200.000 zugelassene Anfragen pro Sekunde zu bedienen, mit etwa 16 MiB residentem Speicher unter dieser Last und einem p99.9 von 62 µs bei 100.000 Anfragen pro Sekunde. Das sind die Zahlen eines Hosts über Loopback, und wir bieten sie als Beleg dafür an, dass der Entwurf trägt, nicht als Rangfolge. Wir haben unser eigenes Konsensprotokoll entfernt und sagen das offen; dafür ist das System kleiner, und seine Datenebene hat keine Abhängigkeit von Dritten. Vieles bleibt zu messen, und wir haben versucht, genau zu sagen, was.

13 Quellen und Reproduktion

Jede Zahl in diesem Papier stammt aus einer eingecheckten Ergebnisdatei des Sibuna-Repositorys, aus einem Dokument einer Sibuna Discussion (SID) oder aus dem Sibuna-Handbuch, und jede Tabelle und jede Fußnote nennt ihre Quelle. Die Ergebnisdateien schreiben `cargo xtask bench primitives`, `cargo xtask bench product` und `cargo xtask bench latency`, und `benchmarks/results/README.md` nennt die Befehle, die CPU-Mengen und die Herkunft jeder Datei.

- `benchmarks/results/primitives-latest.md`, `product-latest.md`, `latency-latest.md` und ihre `.json`-Dateien; `console-8-dashboards-2026-10-15.md`.
- SID 0002 (Grundarchitektur und ihre Revisionsnotizen), SID 0004 (die Betriebsform Shield), SID 0006 (mathematische Grundlagen), SID 0007 (die Konsole), SID 0010 (Workspace, Crates und Abhängigkeitsrichtlinie), SID 0011 (Speicherung über sqlx), SID 0012 (TLS der Konsole), SID 0013 (Speicher der Konsole) und SID 0014 (JA4-Fingerabdrücke vom Eingangsproxy), in `docs/sid/records/`.
- Das Sibuna-Handbuch, `docs/book/`: Teil I (die Kosten einer Anfrage), Teil II (Stand der Technik), Teil III (Kryptographie), Teil IV (das Protokoll), Teil V (Technik ohne Allokation), Teil VI (Automaten und die semantische Firewall), Teil VIII (empirische Bewertung) und Teil IX (Betrieb).
- Die Primärliteratur: Back, Hashcash (1997); Cohen und Pietrzak, Simple Proofs of Sequential Work (EUROCRYPT 2018); Blocki, Lee und Zhou, On the Security of Proofs of Sequential Work in a Post-Quantum World (ITC 2021); Aho und Corasick, Efficient String Matching (1975); Celis, Robin Hood Hashing (1986); der Generic Cell Rate Algorithm des ATM Forum (1996).

Sibuna-Whitepaper | November 2026 | Release 1.0.0-beta.3
Spezifikation, Quellcode und Benchmarks: https://github.com/insanai/sibuna